VAGABOND

The Design and Analysis of a

Temporal Object Database Management System

Kjetil Nørvåg

Department of Computer and Information Science
Norwegian University of Science and Technology

2000

Abstract

Storage costs are rapidly decreasing, making it feasible to store larger amounts of data in databases. However, the increase in disk performance is much lower than the increase in memory and CPU performance, and we have an increasing secondary storage access bottleneck. Even though this is not a new situation, the advent of very large main memory has made new storage approaches possible.

In most current database systems, data is updated in-place. To support recovery and increase performance, write-ahead logging is used. This logging defers the in-place updates. However, sooner or later, the updates have to be applied to the database. This often results in non-sequential writing of lots of pages, creating a write bottleneck. To avoid this, another approach is to eliminate the database completely, and use a *log-only* approach, similar to the approach used in *log structured file systems*. The log is written contiguously to the disk, in a no-overwrite way, in large blocks.

This thesis presents the architecture and design of Vagabond, a temporal object database management system (ODBMS) based on the log-only principle. Solutions to problems regarding temporal data management, fast recovery, efficient management of large objects, dynamic reclustering, and dynamic tuning of system parameters are provided. This includes a new index structure for indexing temporal objects, persistent caching of index entries to solve the object indexing bottleneck, algorithms for transaction management, and declustering strategies to be used in a parallel temporal ODBMS.

In order to compare the log-only approach with the traditional in-place update approach, analytical cost models are used to study the performance of the approaches. The analysis shows that with the workloads we expect to be typical for future ODBMSs, the log-only approach is highly competitive with the traditional in-place update approach.

Many of the ideas presented in this thesis are also useful outside the log-only context. In papers included as appendixes, we show how the ideas can be applied to temporal ODBMSs based on traditional in-place updating techniques.

Preface

This is a doctoral thesis submitted to the Norwegian University of Science and Technology for the doctoral degree "doktor ingeniør". This work has been carried out at the Database Systems Group, Department of Computer and Information Science, at the Norwegian University of Science and Technology, under the supervision of Prof. Kjell Bratbergsengen. The doctoral study was funded by the Norwegian Research Council (NFR).

Acknowledgments

First of all I want to thank my advisor Kjell Bratbergsengen for guidance and many good ideas throughout the work towards my doctoral degree.

During the years I have worked on this thesis many people have helped me reach my goal. In particular, I would like to thank Olav Sandstå for insightful discussions, lots of valuable feedback, and for taking the time to proofread this thesis as well as my papers. I also want to thank the other members of the Database Systems group for providing a good environment for doctoral students. During the last two years I have been employed as a lecturer in the Group for Computer Architecture and Design, and I want to thank Lasse Natvig and Pauline Haddow for providing me the opportunity to finish this thesis. I am also grateful to the department's always friendly and helpful administrative staff.

I would also like to thank Prof. Malcolm Atkinson for many valuable comments on the thesis.

Finally, I thank my family, who have always encouraged me. This support has been of great value.

Contents

I Overview **1**

1 Introduction **3**
1.1 Application Areas 4
1.2 The Need for a New Architecture 7
1.3 Outline of the Thesis 8

2 Object Database Management Systems **11**
2.1 What is an Object Database System? 11
2.2 The ODMG Standard 13
2.3 Object Database Systems 14
2.4 Summary 16

3 Design Issues **17**
3.1 Object Identifiers 17
3.2 Object Storage Structure 18
3.3 Object Clustering 19
3.4 Client/Server Architectures 20
3.5 Method Execution 21
3.6 Data Granularity 23
3.7 Buffer Management 25
3.8 Indexing 27
3.9 Swizzling 28
3.10 Query Processing 28
3.11 Parallel ODBMSs 29
3.12 Summary 33

4 Temporal Database Systems **35**
4.1 What is a Temporal DBMS? 35
4.2 Data Models 35
4.3 Temporal Queries and Query Languages 37
4.4 Programming Language Bindings 38
4.5 Vacuuming 41
4.6 Implementation Issues 41
4.7 Temporal ODBMSs 43
4.8 Summary 43

5 Log-Only Database Management Systems **45**
5.1 The Log-Only Approach . . . 45
5.2 Advantages of a Log-Only Approach . . . 48
5.3 Alternative Realizations . . . 50
5.4 Systems Based on Log-Only Related Techniques . . . 51
5.5 Summary . . . 54

II The Design of Vagabond **55**

6 An Overview of Vagabond **57**
6.1 Server Architecture . . . 57
6.2 Objects in Vagabond . . . 60
6.3 Read and Write Efficiency Issues . . . 63
6.4 Parallelism and Distribution in Vagabond . . . 65
6.5 Summary . . . 66

7 Reducing the Data Transfer Volume **67**
7.1 Signatures . . . 67
7.2 Object Compression . . . 72
7.3 Summary . . . 72

8 Object-Identifier Indexing **73**
8.1 Contents and Structure of the OID Index . . . 73
8.2 Declustering . . . 78
8.3 Temporal OID Indexing . . . 79
8.4 VTOIDX: The Vagabond Temporal OID Index . . . 85
8.5 Large Objects . . . 92
8.6 Reducing the OIDX Access Costs . . . 98
8.7 Log-Based vs. In-Place Updated OIDX . . . 100
8.8 Object References and Remote Objects . . . 102
8.9 Tertiary Storage Indexing . . . 102
8.10 Summary . . . 103

9 The Persistent Cache **105**
9.1 Introduction . . . 105
9.2 PCache Organization . . . 107
9.3 LRU Management . . . 107
9.4 Update Operations . . . 108
9.5 Object Creations . . . 108
9.6 Read Operations . . . 108
9.7 PCache-to-TIDX Writeback . . . 109
9.8 PCache and TIDX on Tertiary Storage . . . 110
9.9 Summary . . . 110

10 Large Objects in Vagabond **111**
10.1 Why Special Object Handlers? . . . 111
10.2 Special Object Handler Services . . . 113
10.3 Examples of Special Objects . . . 114
10.4 Summary . . . 115

11 Temporal Object Declustering **117**
11.1 Introduction . . . 117
11.2 Related Work . . . 119
11.3 Object Declustering in Server Groups . . . 120
11.4 Object Declustering in Distributed Systems . . . 126
11.5 Summary . . . 126

12 Log-Only Database Operations **129**
12.1 Introduction . . . 129
12.2 Object Operations . . . 131
12.3 Transaction Management . . . 140
12.4 Controlled Shutdown and Restart . . . 143
12.5 Recovery . . . 143
12.6 Vacuuming . . . 145
12.7 Segment Cleaning . . . 146
12.8 Schema Management . . . 152
12.9 Object Migration . . . 152
12.10 Backup . . . 153
12.11 Query Processing . . . 153
12.12 Volume Management . . . 153
12.13 Transparent Use of Tertiary Storage . . . 154
12.14 Node Operations . . . 154
12.15 Summary . . . 155

13 Physical Data Structures **157**
13.1 Data Volume Structures . . . 157
13.2 Memory Data Structures . . . 162
13.3 Summary . . . 170

III Analysis and Conclusions **171**

14 Analysis of the Log-Only Approach **173**
14.1 Analytical Modeling . . . 173
14.2 Cost Model . . . 174
14.3 Object Access Model . . . 175
14.4 The BDD LRU Buffer Model . . . 176
14.5 Assumptions Behind the ODBMS Models . . . 188
14.6 Analytical Modeling of an IPU-ODBMS . . . 191
14.7 Analytical Modeling of an LO-ODBMS . . . 195
14.8 A Comparison of Performance . . . 196

14.9 OIDX Costs 205
14.10 Summary 206

15 A Comparison of Declustering Strategies 207
15.1 Cost Model 207
15.2 Analysis 213
15.3 Summary 217

16 Conclusions and Further Work 219
16.1 Is Vagabond a Suitable Solution? 219
16.2 Contributions and Publications 221
16.3 Criticism 222
16.4 Future Work 223

IV Appendixes and Additional Papers 225

A SCCC'96 227

B BNCOD15 239

C DEXA'98 257

D FODO'98 271

E VLDB'99 283

F ADBIS'99 297

G Validation of the Index Buffer Model 313
G.1 The Index Buffer Simulator 313
G.2 Results 314
G.3 Related Work 315
G.4 Conclusions 315

H Abbreviations 317

References 319

List of Tables

2.1 ODBMSs and storage managers with language binding. 15
2.2 Object Relational Database Systems. 16

8.1 Contents and size of fields in the object descriptor. 75

13.1 Device information block. 158
13.2 Checkpoint block. 159
13.3 Volume device table. 159
13.4 Segment structure. 161
13.5 Entry in the resident small object table. 167
13.6 Entry in the segment status table. 170

14.1 Partition sizes and partition access probabilities. 175
14.2 Partition sizes and partition access probabilities for three books 176
14.3 Summary of system parameters and functions used in the models. 189
14.4 IPU-ODBMS specific parameters and functions. 191
14.5 LO-ODBMS specific parameters and functions. 195

15.1 Summary of system parameters and functions. 208

G.1 Partition sets used in the index buffer validation. 313

List of Figures

3.1 OID in Objectivity/DB . . . 18
3.2 Client/server architectures. . . . 20
3.3 Alternative parallel architectures. . . . 30

5.1 Disk volume structure. . . . 46
5.2 Data and index in a log-only ODBMS. . . . 46
5.3 Segment states. . . . 47
5.4 POSTGRES file. . . . 51
5.5 POSTGRES page. . . . 51
5.6 POSTGRES record. . . . 52
5.7 LSM with four components. . . . 53

6.1 The Vagabond server. . . . 58
6.2 Class descriptor (CDO). . . . 61
6.3 Vagabond system architecture. . . . 65

8.1 OIDX with containers. . . . 74
8.2 Distributed system, with server groups and servers. . . . 79
8.3 One-index structure using the concatenation of OID and commit time . . . 81
8.4 One-index structure, with version linking. . . . 83
8.5 Nested ST indexing . . . 84
8.6 The Vagabond temporal OID index. . . . 85
8.7 Large object, linked list approach . . . 93
8.8 Versioned large object, linked-list approach . . . 93
8.9 Large object, pointer array approach . . . 94
8.10 Large object, with subobject-index . . . 95
8.11 Large object in EXODUS . . . 95
8.12 Versioned large object, single-level subobject index . . . 96
8.13 Versioned large object, multi-level subobject index . . . 97
8.14 Contents and size of fields in the subobject descriptor (SOD). . . . 97
8.15 Index page buffer and OD Cache. . . . 98
8.16 Hybrid OIDX . . . 101

9.1 Overview of the TIDX, PCache, and index-related main-memory buffers. . . . 106

11.1 Object versions versus time . . . 118
11.2 Two declustering strategies . . . 118
11.3 OID-based declustering . . . 121

11.4 *TIME* declustering . . . 122
11.5 *OID-TIME* declustering . . . 125

12.1 Example of log writing . . . 130
12.2 Long transaction. . . . 132
12.3 2-phase commit . . . 141

13.1 Data volume . . . 157
13.2 Important memory buffers . . . 163
13.3 OD cache . . . 165
13.4 Block numbering in BSD-LFS . . . 168
13.5 Large granularity buffer indexing . . . 169

14.1 Logical access partitions. . . . 174
14.2 TSIM architecture. . . . 178
14.3 OD cache hit rate for read only accesses. . . . 181
14.4 OD cache hit rate with mixed workload. . . . 182
14.5 OD cache hit rate with mixed workload. . . . 183
14.6 Deviation between simulation and the DCOMP model . . . 183
14.7 Deviation, different write rates . . . 184
14.8 Deviation, different object create rates. . . . 185
14.9 Deviation, different amounts of temporal data . . . 186
14.10 Deviation, different amounts of dirty data in the OD cache . . . 187
14.11 Throughput during different operational phases. . . . 190
14.12 Object access cost. . . . 198
14.13 Speedup with different object sizes S_{obj}. . . . 199
14.14 Speedup with different update ratios P_{write}. . . . 200
14.15 Speedup with different clustering factors C. . . . 201
14.16 Speedup from using compression in an LO-ODBMS . . . 202
14.17 Speedup with different checkpoint-interval lengths N_{CP}. . . . 203
14.18 Speedup with disk read-ahead. . . . 204

15.1 Cost with different update rates . . . 214
15.2 Cost with different object sizes . . . 215
15.3 Cost with different read mix . . . 216

G.1 Overall buffer hit probability with different buffer sizes . . . 314
G.2 Relative deviation . . . 314
G.3 Relative deviation, no traverse strategy is used . . . 315

Part I

Overview

Chapter 1

Introduction

The recent years have brought computers into almost every office, and this availability of powerful computers, connected in global networks, has made it possible to utilize powerful data management systems in new application areas. The increasing performance and storage capacity, combined with a decreasing price, have made it possible to realize applications that were previously too heavy for medium- and low-cost computers. However, high performance and storage capacity is not enough. We need support software, including database management systems, operating systems, and compilers, that are able to benefit from the advances in hardware. This often means rethinking previous solutions, similar to what was done in the hardware world with the introduction of the RISC concept.

In this thesis, we concentrate on database management systems (DBMS), quite likely to be the bottleneck in many future systems. The first step in the process of rethinking old solutions has already been done, with the advent of object database management system (ODBMS).[1] While relational database management systems (RDBMSs) have good performance for many of the traditional application areas, new applications demand more than traditional RDBMSs can deliver. The increased modeling power and removal of the language impedance mismatch in ODBMSs, have made integration between application programs easier, and in many cases helped to increase the performance of the applications.

Traditionally, data (objects/tuples) have lived in an artificial, modeled world, after being inserted into the database. This creates a mismatch in many ways similar to the language impedance mismatch in RDBMSs. What we would like, is DBMSs supporting a world more similar to our own, which includes *time and space*. This is not at all a new observation, especially the aspect of temporal database management has been an active research area for many years. However, current database architectures, which are adequate for yesterday's applications, might have problems coping with tomorrow's application. In this thesis, we will reconsider some of what is "established truth", and propose a new architecture, the *Vagabond Temporal Object Database Management System*, which should be more suitable for tomorrows applications.

Before we finish this section, it is very important to emphasize that some of the ideas in this thesis are not new. However, many of the ideas did not have a supporting framework when they were proposed. Hence, many of the ideas are now forgotten. One notable exception, is some of the ideas from the POSTGRES system. POSTGRES included many novel ideas, which, because they where incorporated into a system, managed to survive. Unfortunately, POSTGRES was in many ways too early, and even though many of the elements of POSTGRES survived into current object-relational systems, some of the ideas we will concentrate on in this thesis, like the no-overwrite strategy, and

[1] The term *object-oriented database management system* (OODBMS) was previously used, but now the more precise term *object database management system (ODBMS)* has gained acceptance.

keeping previous versions, have later had little attention in DBMS research.

In the rest of this chapter, we will motivate the work that will be presented in the rest of this thesis. In Section 1.1 we describe some application areas that have only limited support in existing database system. Based on this discussion, we summarize some of the problems and shortcomings of current systems in Section 1.2, and outline assumptions and features that motivated the design of the Vagabond system, which will be described in detail throughout the rest of the thesis. In Section 1.3, we outline the structure of the rest of the thesis.

1.1 Application Areas

We can categorize application areas into *existing* application areas, and *emerging* application areas. Existing application areas include the traditional database areas, for example transaction processing applications, well suited for RDBMSs. They also include application areas where application specific DBMSs or file systems have been used earlier, because existing general purpose DBMSs can not handle the performance constraints. Emerging application areas includes both new application areas, that are emerging as a response to the increased computer performance in general, and application areas that are a response to other technologies, for example the World Wide Web.

We will in this section first describe some examples of existing applications where DBMSs until recently have been a potential performance bottleneck:

- Geographical information systems.
- Scientific and statistical databases.
- Multimedia systems.
- PACS (picture archiving and communications systems).

Next, we will describe some applications where increased database support will be needed in order to deliver the desired performance:

- Temporal DBMSs.
- Semistructured data management/XML.

Geographical Information Systems. A geographical information system (GIS) is a system for management of geographical data, i.e., *data which describes phenomena directly or indirectly associated with a location (and possibly time and orientation as well) relative to the surface of the Earth* [34].

Earlier, GIS employed the DBMS (usually a RDBMS) to manage the fact data[2] only, but used proprietary file management systems to take care of geometrical and topological data. The main reason for this, was that most RDBMSs did not support sequences ("ordered sets"), and retrieving polygons from relations was (and still is) prohibitively expensive. This is unfortunate, because the file management systems tend to be single-user, and there are no transactional access control as in DBMSs. Recently, GIS have been built by extending database systems with spatial data types. However, these ad-hoc solutions do not really address the main problem, the data model: concepts are simple, the data

[2]Fact data is data describing the objects, e.g., the name of a road, but not the "road object" itself.

type system is weak, data have to be normalized in first normal forms while hierarchical structures are needed, semantic links are lost and need to be rebuilt through semantic constraints, and the data access system is very expensive because of joins [51].

With its increased modeling power, ODBMSs are ideal for GIS applications. They support complex objects and relationships efficiently. However, some ODBMSs do not have sufficient support for large objects, and not all ODBMSs have a scalable architecture.

Scientific and Statistical Databases. Scientific and statistical databases (SSDBs), for example survey data and data from physical experiments, have many characteristics in common, which makes it practical to consider them together:

- The size of the databases are usually *very large.*
- The update frequency is often *very low*. The reason for this, is that the primary purpose of an SSDB is to collect data for future reference and analysis.
- Bulk loading is frequently used to insert data into the database.
- The read/write ratio can be low. Because of the size of the database, summary data is often used instead of the whole database in queries.
- Data in both scientific and in statistical databases are eventually statistically analyzed.
- Complex relationships exist between data. For example, experiments not only carry result data, but also configuration and environmental data.
- Multidimensional data is frequent.
- Data is often sparse, i.e., many attributes have a NULL value.

The size of SSDBs pose a problem for many DBMSs, and the complex relationships make a data model with high modeling power desirable. Traditional systems do not support multidimensional data well, and in the case of sparse data, efficient support for compression is necessary. This does not only include support for compression and decompression itself, but also efficient access and manipulation of compressed data.

In analysis, statistical operators are needed. These are not included in traditional systems. Another important feature in practical SSDBs, is bulk loading, which few systems handle well.

Database research and development is highly market driven, and until very recently, the active research in this area was very limited. This has changed dramatically the last few years, with the increased interest in data warehousing/OLAP, which has many similarities with SSDBs.

Multimedia Systems. Multimedia data management differs from traditional data management in several ways:

- Large objects, for example images and videos, are common. In general, there are sufficiently many large objects stored in such a database to make the total database size large as well.
- New and complex data types.
- New types of queries. One example is query by contents, another is queries on image characteristics (for example on image histograms).

- Isochronous retrieval: In the case of dynamic data, like video, data is to be delivered in pieces of the object at regular time intervals, and not the whole object at once. The scheduling of this data delivery is complicated, as is witnessed from dedicated video servers, which do not have to care about the other database aspects.

Even though some vendors define all databases capable of storing large objects as multimedia DBMS, the fact is that current DBMSs have only limited support for multimedia data, and this is particularly true for isochronous delivery of the data stored in the database.

Picture Archiving and Communications Systems. Picture archiving and communications systems (PACS) will be an important part of tomorrow's health care. In the future different kinds of data will be stored in such systems, but currently most systems concentrate on storage of pictures, for example X-ray pictures. The pictures stored in these systems are required to have a high resolution, and with the number of pictures to be stored in such a system the database size will be very large. The historical data in a PACS system will be very infrequently accessed, and can be stored in tertiary storage.

Temporal Database Management Systems. A temporal DBMS is a DBMS that supports some aspects of time. Informally, this means that an object (or a tuple) is associated with time, and that the object can exist in several versions, each version being valid in a certain time interval. An example is the salary of a person. If the salary is represented as a temporal object, a new object version is created every time the salary is changed. In a temporal DBMS, this versioning, related to time, is supported and maintained by the system, which also provides support for querying the temporal data.

The temporal aspect exists in most real life databases, where some or all of the data is associated with some aspect of time. Examples include:

- Accounting: What bills were sent out and when, and what payments were received and when.
- GIS: The geography, such as rivers, and the existence, shape and size of objects such as houses and roads, change over time.
- Stock marked data.
- Patient records.
- Personnel information, including salary histories.
- Airline reservation systems.
- In scientific DBMSs, timestamping of data is important, for example for data from an experiment that is repeated several times.

We will give a more detailed introduction to temporal DBMSs in Chapter 4.

Semistructured Data Management/XML. A very active research area at the moment, is *semistructured data management.* Semistructured data is data where the information that is normally associated with a schema, is contained *within the data*. In some forms of semistructured data there are no separate schemas, in others it exists, but only places loose constraints of the data [35].

The main reasons for the heavy interest in semistructured data are its application in data exchange/data integration, and the large amount of semistructured data available on the Web. Research

in semistructured data management has recently been concentrated on the theory of semistructured data; models, query languages, but less on physical management. Several approaches have been taken in incorporating the ideas on top of traditional ODBMSs, for example on O_2 [2], but only a few systems have been specially designed for semistructured data. One example of such a system is Lore [138].

ODBMSs are very appropriate for semistructured data management, as their underlying model has many similarities with most semistructured data models, for example the Object Exchange Model (OEM) [3]. However, some features not provided by most existing ODBMSs are desired. One example is query languages suitable for semistructured data, and appropriate optimization techniques. If these DBMSs should deliver reasonable performance, new indexing techniques are also needed, including full text indexes.

As pointed out by Abiteboul [1], we are often more concerned by querying the recent changes in some data source than in examining the entire source. Support for temporal data in the storage layer would facilitate this, and this can also be useful in distributed DBMSs where data is exchanged in bulk at regular intervals.

1.2 The Need for a New Architecture

Based on the discussion in the previous section, we have identified some features that should be supported by future database systems:

- Temporal data and operations on these.
- Large objects and flexible partitioning of large objects.
- Isochronous delivery of data.
- Queries on large data sets.
- In applications with low read/write ratio, it should be possible to use this characteristic to increase performance.
- Full text indexing.
- Multidimensional data.
- Efficient storage of sparse data (for example by the use of data compression).
- Dynamic clustering and dynamic tuning of system parameters.

Until now, no single existing system has supported all these features. Ad-hoc solutions exist for some of the features, but these are often not scalable, or will not work well together with support for the other features. We believe that future systems should efficiently support these features, in *one integrated system*. In this thesis, we show how this can be done, through the design of the temporal ODBMS Vagabond.[3] Vagabond is designed to support the listed features, with a philosophy based on the following assumptions:

[3]From *Webster's Encyclopedic Unabridged Dictionary*: Vagabond: "a person, usually without a permanent home, who wanders from place to place; nomad". Quite similar to our objects!

1. Although many of the current problems might be handled by future main-memory database management systems (MMDBMSs), there are many problems (and more will appear, as the computers become powerful enough to solve them) that require the management of larger amounts of data than can be handled by a MMDBMS alone. However, the increasing amounts of main memory should be utilized as far as possible in order to reduce time consuming secondary storage accesses.

2. *The main bottleneck* in a DBMS for large databases is still secondary storage access. In a DBMS, most accesses to data are read operations. Consequently, database systems have been *read-optimized*. However, as main-memory capacity increases, we expect that the amount of disk-write operations relative to disk-read operations will increase (most read operations can be satisfied from the main-memory buffer). This calls for a focus on *write-optimized* DBMSs.

3. To provide the necessary computing power *and* data bandwidth, a parallel architecture is necessary. A shared-everything approach is not scalable, so our primary interest is in ODBMSs based on shared-nothing multicomputers. With the advent of high performance computers, and high speed networks, we expect multicomputers based on commodity workstations/servers and networks to be most cost effective.

4. In most application areas, there is a need for increased data bandwidth, and not only increased transaction throughput (although these points are related). This is especially important for emerging application areas such as multimedia and supercomputing applications, which have earlier used file systems.

5. Even though set-based queries have been a neglected feature in most ODBMSs, we expect it to be as important in the future for ODBMSs as it has been previously for RDBMSs. The popularity of the hybrid object-relational systems justifies this assumption.

6. Distributed information systems are becoming increasingly common, and they should be supported in a way that facilitates both efficient support for distribution, *and* efficient execution of local queries and operations.

1.3 Outline of the Thesis

The thesis is logically divided into four parts. The first part, Chapter 2 to 5, is mainly an introduction to ODBMSs and ODBMS implementation issues, temporal DBMS, and the log-only approach.

- *Chapter 2* describes the most important features of ODBMSs, gives an overview of the ODMG standard, and outlines the history of ODBMSs.
- *Chapter 3* discusses design issues in ODBMSs.
- *Chapter 4* gives an introduction to temporal DBMSs in general.
- *Chapter 5* gives an introduction to log-only DBMS, and a short overview of previous systems based on the log-only approach.

In the second part, the architecture of the Vagabond log-only DBMS and the most important algorithms are described in detail.

- *Chapter 6* describes the architecture of the Vagabond temporal ODBMS.
- *Chapter 7* discusses two techniques for reducing the data transfer volume: signatures and object compression.
- *Chapter 8* studies the problems of indexing object identifiers (OIDs) in a temporal ODBMS, and proposes a new indexing structure suitable for this task.
- *Chapter 9* introduces a novel structure called the *Persistent Cache*, which reduces the OID indexing cost.
- *Chapter 10* gives a more detailed description of large objects and their use in Vagabond.
- *Chapter 11* discusses object declustering in parallel and distributed temporal ODBMSs.
- *Chapter 12* describes the most important operations in Vagabond.
- *Chapter 13* describes the most important physical data structures in Vagabond.

In the third part, log-only database systems are compared analytically with traditional in-place updating ODBMSs, and we conclude the thesis.

- *Chapter 14* contains analytical models of a log-only ODBMS and an in-place update ODBMS, and uses these models to compare the hypothetical performance of the two approaches.
- *Chapter 15* contains a qualitative analysis of the declustering strategies discussed in Chapter 11.
- *Chapter 16* concludes the thesis and outlines directions for further research.

The fourth part, Appendix A to F, is a compilation of papers that discuss issues not covered in detail by the main part of the thesis. The four last papers show how the results of the main part of the thesis are also applicable for temporal ODBMSs based on traditional techniques.

- The paper "Aggregate and Grouping Functions in Object-Oriented Databases", presented at SCCC'96, is included in Appendix A.
- The paper "Improved and Optimized Partitioning Techniques in Database Query Processing", presented at BNCOD'97, is included in Appendix B.
- The paper "An Analytical Study of Object Identifier Indexing", presented at DEXA'98, is included in Appendix C.
- The paper "Optimizing OID Indexing Cost in Temporal Object-Oriented Database Systems", presented at FODO'98, is included in Appendix D.
- The paper "The Persistent Cache: Improving OID Indexing in Temporal Object-Oriented Database Systems", presented at VLDB'99, is included in Appendix E.
- The paper "Efficient Use of Signatures in Object-Oriented Database Systems", presented at ADBIS'99, is included in Appendix F.

In Appendix G we present a validation of the index buffer model used in the papers in Appendix C, D, E, and F. In addition, a list of abbreviations used in this thesis is provided in Appendix H.

Chapter 2

Object Database Management Systems

Relational database management systems (RDBMS) have revolutionized database management during the last 20 years. Important reasons for the success are SQL, and the ability to efficiently perform queries over large amounts of data. However, RDBMSs are based on a simple data model. Even though this gives high performance for many typical data-retrieval applications, the result can often be very low performance in applications managing data with complex relationships. For example, until very recently it was impossible to design a GIS systems based on traditional RDBMS technology.[1] In addition, in many applications with high transactions rates, systems based on hierarchical and network data models have continued to be used.

For some application areas, a more complex data model and focus on data manipulation, rather than data retrieval, is desired. Typical examples of such systems have been GIS, CAD, software development systems and more recently also Web databases. Many applications also need to do complex operations on the data. In a typical RDBMS, this has to be done by accessing the database from the application program by using database commands embedded in some general programming language. This *language impedance mismatch* is costly and inefficient.

Object database management systems (ODBMS), previously called object-oriented database management systems, emerged as an answer to the shortcomings of previous models and systems. The rest of this chapter will give an introduction to ODBMS, and we will start by defining the term *object database management system (ODBMS)* in the next section. An overview of the world of ODBMS would not be complete without an overview of the contents of the ODMG standard, which is given in Section 2.2. To set our work into perspective, we briefly outline the history of ODBMSs in Section 2.3, from the first approaches in persistent programming languages, via storage managers, to today's commercially available ODBMSs. We also give a brief overview of object-relational database management systems (ORDBMS).

2.1 What is an Object Database System?

As is obvious from the ODBMS research prototypes and commercially available ODBMSs, the design space for an ODBMS is much larger than for RDBMSs. However, there are some features and characteristics shared by most of them, initially described in *The Object-Oriented Database System Manifesto* by Atkinson et al. [6]. We will now summarize the most important features, separated into language related features (the OO part), and the database features (the DB part).

[1] A notable exception is Techra [204], which over a decade ago included support for GIS data management, including sequences.

Language features:

- Complex objects. The ability to build complex objects from simpler ones by applying constructors to them. Complex object constructors include tuples, sets, bags, lists, and arrays.
- Object identity. All objects have a system managed identity that is independent of the value of the object. The identity is assigned by the system, can not be altered by the user, and remains the same even when the value of the object changes.
- Encapsulation. Encapsulation is used to distinguish between the specification and the implementation of an operation. No operations, outside those specified in the interface, can be performed. This restriction holds for both update and retrieval operations.
- Types or classes. Types or classes should be supported. The ODMG standard encapsulates both, and the language binding used decides to what extent these concepts are supported.
- Class or type hierarchies. Inheritance is a powerful modeling tool, because it gives a concise and precise description of the world, and it helps in factoring out shared specifications and implementations in applications.
- Overriding, overloading and late binding. This is the concept of having several implementations of an operation, for each of the types. Which implementation to use, is decided at run-time, *late binding*.
- Computational completeness. To avoid the language impedance mismatch, the data manipulation language should be computationally complete.
- Persistence. Persistence is the ability of the programmer to have her/his data survive the execution of a process, in order to eventually reuse the data in another process. Persistence should be orthogonal, i.e., each object, independent of its type, is allowed to become persistent as such (i.e., without explicit translation). It should also be implicit: the user should not have to explicitly move or copy data to make it persistent.

Database features:

- Secondary storage management. Database mechanisms as index management, data clustering, data buffering, access path selection and query optimization should be invisible to the user: they are simply performance features. There should be a clear independence between the logical and the physical level of the system.
- Concurrency and recovery. The system should offer the same level of service as traditional database systems, i.e., atomicity and controlled sharing when multiple users access and update data. The same applies to recovery, in case of hardware or software failures, the system should recover, i.e., bring itself back to a consistent state.
- Ad-hoc query facility. The system should provide functionality of an ad-hoc query language, though not necessarily as an own query language. This is probably the feature where current ODBMSs differ most. While some systems, like O_2, offer a SQL like language (OQL), with query optimization similar to RDBMSs, other systems only provide primitive scan operations.

The *Manifesto* also lists some additional features, the most important being support for *distribution*, *design transactions* ("long" transactions) and versioning. These features are not mandatory to make a database system an object-oriented system, but features that are desired in many of the typical ODBMS applications. Thus, these features are supported to some extent in most commercial ODBMSs.

2.2 The ODMG Standard

ODBMS is now a relative mature technology, and commercial ODBMSs have proved to be competitive with RDBMSs in many application areas, and superior in others. They have been able to deliver high performance and provide high availability. Still, they have not managed to seriously threaten the traditional RDBMSs.

There are several reason why the ODBMS market segment is still small, but one important factor has been lack of standardization. One important reason for the success of the RDBMSs, is the common data model and the common data specification and manipulation language. This was realized by the ODBMS vendors in the early 90's, and the *Object Database Management Group (ODMG)* was formed in 1991 to develop and promote standards for object storage. The participants of the ODMG includes representatives from all major ODBMS vendors. Recently, the focus of the ODMG have been broadened, and the name changed to the *Object Data Management Group*.

2.2.1 The Components of the ODMG Standard

The components of the ODMG standard [43] are built upon the ODMG object model, which is a superset of the OMG object model. The specification covers three areas:

- Object definition language (ODL).
- Object query language (OQL).
- Language bindings.

Object Definition Language. ODL is in fact a syntax of the object model, and is a superset of OMG's IDL. It can be used to define a database schema in a programming language independent manner in terms of object type, attributes, relationships and operations. The resulting schema can be moved from one database to another. The schema of an application can be translated to declarations in different programming languages. These schemas can be included in the application code.

Object Query Language. OQL is a declarative query language, and is a superset of the part of SQL that deals with database queries. It includes support for object sets and structures, and has object extensions to support object identity, complex objects, path expressions, operation invocation, and inheritance.

Language Bindings. ODBMSs are accessed through languages with support for persistent objects, usually extensions of existing general purpose programming languages. The ODMG language bindings define extensions to the languages to support and integrate OQL, navigation and transactions. Currently, language bindings for C++, Java and Smalltalk have been standardized.

2.2.2 The ODMG Standard in Practice

The ODBMS vendors have been slow at adopting the ODMG standard, and unfortunately, they currently seem even less eager to do so. Vendors can claim compliance with one or more components of the ODMG standard, i.e. one or several of the C++/Java/Smalltalk language bindings, and OQL. Even though many vendors claim that their systems are ODMG compliant, the lack of certification procedures is a problem. The only vendor close to supporting the whole standard was O_2, which is no surprise, as the standard itself, especially OQL, borrowed heavily from O_2. However, the O_2 is no longer on the market.

Currently, there seems to be little interest in continued work on the ODMG object model, OQL, and the language bindings. The model is only used in the ODMG specification itself, the ODBMS vendors prefer to use their proprietary C++ language bindings, and OQL has only limited support. Most of the interest at the moment is in the Java binding and object/relational mappings, and it is very likely that future work will be in these directions.

2.3 Object Database Systems

To set our work into perspective, we briefly summarize previous work on ODBMSs. We have summarized all implemented systems we are aware of in Table 2.1. This summary is provided for two reasons. First of all, we want to show that ODBMSs have been an active research area, and still is. Second, we provide the summary with references, to make it easier for others to probe earlier works, as we are not aware of any other published summary or survey trying to cover the implemented systems. In this summary, we classify the systems in three groups:

- Early approaches.
- ODBMSs and storage managers with language binding.
- ORDBMSs.

For the commercial systems, the publications cited do not necessarily represent descriptions of the current versions of the commercial systems. For information on current versions of the systems and their features, the reader is encouraged to visit the Web sites of the respective ODBMS companies.

2.3.1 Early Approaches to Persistent Programming Languages

Traditionally, users and applications have communicated with the database system through special data definition languages (DDL) and data manipulation languages (DML). Operations on tuples have been done with some predefined functions. If more advanced operations were desired, a general purpose language with embedded database language functions were used. With this approach, you get a language impedance mismatch.

To avoid the language impedance mismatch, new systems was developed. In these systems, there were no distinction between database (persistent) and no-database (transient) data, the same language is used for both. The first such systems were ASTRAL [32, 33] and PASCAL/R [181]. Later, other systems followed, for example PS-algol [49]. In the next phase of the evolution, persistent versions of Smalltalk and persistent C++ became popular, used in combination with the storage managers summarized in the next section.

Name	References	Name	References
AGNA	[145]	O_2	[7, 53, 165]
Amadeus	[80, 203]	Objectivity	[167]
BeSS	[17]	ObjectStore	[118, 171]
Bubba	[28]	OBST	[42]
Cricket	[186]	ODB-II	[168]
Dalí[2]	[99]	ODE	[71]
DASDBS	[180]	ONTOS	
Eiffel**	[137]	ORION	[112]
Encore	[89]	OSAM*.KBMS/P	[201]
EOS	[81]	PJama	[175]
EXODUS	[38]	Poet	
EyeDB		PPOST	[22]
GemStone	[37, 133]	PRIMA	[74, 75]
ITASCA	[97]	Ptool	[79]
Iris	[68, 215]	QuickStore	[214]
Jasmine	[95]	Shore	[39]
KIOSK	[146]	Texas	[189]
Lumberjack	[92]	Thor	[128, 130]
MATISSE	[134]	Tycoon	[135]
Mneme	[140, 142]	Versant	
Monet	[26, 27]	VODAK	[115]

Table 2.1: ODBMSs and storage managers with language binding.

2.3.2 ODBMSs and Storage Managers with Language Binding

After the first attempts with database programming languages, systems more closely resembling what we today call object database management systems entered the scene. In many of these systems, the focus was more on persistent programming than database management, and as a result, many of these system have only a very primitive query language, if any at all. However: all systems share one important goal, removing the language impedance mismatch.

The number of ODBMSs and storage managers with language binding is quite large, and the implemented systems we are aware of are summarized in Table 2.1. Most of the systems are only research prototypes, but some of them are commercialized ODBMSs: GemStone, Itasca, Jasmine, MATISSE, O_2, Objectivity, ObjectStore, ODB II, ONTOS, Poet, and Versant.

Several systems are marketed as ODBMSs, but are not included in Table 2.1. The reason for not including theses, is that they either lack some of the more important features expected from ODBMSs (they would more correctly be classified as object file managers or indexing tools), or that we have only limited information about the systems. The systems omitted from the summary include ActiveInfo, GOODS/POST++, Jeevan, Neoaccess (NeoLogic), ObjectFile (ObjectFile Ltd.), OOFILE (A.D. Software), Persist (Persist AG), PLOB! (Persistent Lisp OBjects, from University of Hamburg), Tenecit (Totally Objects), and TERSOL (TechKnowledge).

In addition, several systems use one of the systems in Table 2.1 as the storage manager in the system. This includes AllegroStore, which combines ObjectStore with CLOS (Common Lisp Object System), Multicomputer Texas [18] (described in Section 3.11.3), which uses Texas as the storage

Name	References
DB2	[41]
Illustra/Informix Universal Server	
Oracle 8	
POSTGRES	[195, 200, 197, 198]
Starburst	[85, 131]
UniSQL	

Table 2.2: Object Relational Database Systems.

manager in a parallel ODBMS, and Open ODBMS [20] and METU ODBMS [58], which both were developed on top of the Exodus storage manager.

2.3.3 Object-Relational Database Systems

ORDBMSs[3] carry on the relational paradigm. Data is still organized in relations, but the systems offer additional features, including support for more complex data types, and large objects. The most well-known of these are summarized in Table 2.2.

The history of ORDBMSs started with POSTGRES,[4] later commercialized into Illustra/Informix Universal Server. Currently, most major RDBMS vendors have extended their products to support object-relational features. Some ODBMSs, included in Table 2.1, have also been marketed as object-relational, or have features that make it possible to classify them as object-relational, for example MATISSE and ODB-II.

It is possible to implement an ODBMS on top of an ORDBMS backend, and vice versa. Examples are Paradise [55, 173], which uses Shore [39] as its underlying persistent object manager, and several commercial products that offer Java and C++ language bindings on top of ORDBMSs. Based on this observation, one might think that the division of ODBMS and ORDBMS is artificial, and that the ODBMS vs. ORDBMS discussion is more a debate on what interface to make available for users and programmers. It is important to note that this is not the case. While such approaches deliver the functionality, they are in general not efficient and scalable approaches.

2.4 Summary

We have in this chapter described the most important features of ODBMSs, given an overview of the ODMG standard, and provided an overview of previous and existing ODBMSs. Although we have tried to make the overview as complete as possible, we are fully aware that the list is not complete: many projects have been completed without any publication efforts, and new systems are developed and marketed as this thesis is written.

[2] Dalí has now been commercialized, and renamed *Datablitz*.

[3] Object relational database systems were previously called *extended relational database systems*.

[4] A "cleaned up version" of POSTGRES, *PostgreSQL*, is continuously under development by "the public domain community".

Chapter 3

Design Issues

The design of an ODBMS introduces new issues not found in RDBMSs. Each design issue may have alternative solutions, and few have definite answers. Many of them are also highly related, one of the alternatives for one issue can rule out alternatives for other issues. In this chapter, we discuss the most important issues, and provide a background for the description of Vagabond. This chapter also establishes the terminology which will be used in the rest of this thesis.

3.1 Object Identifiers

An object in an ODBMS is uniquely identified by an object identifier (OID). This OID is used as the "key" when retrieving the object from disk. OIDs can be *physical* or *logical*. If physical OIDs are used, the disk blocks where an object resides is given directly by the OID. If logical OIDs are used, it is necessary to use an OIDX index (OIDX) to map from a logical OID to a physical location. Most of the early ODBMSs and storage managers used physical OIDs because of its performance benefits, and many of the commercial ODBMSs still do. However, using physical OIDs have major drawbacks: relocation and migration of objects are more difficult, which in turn makes schema changes and reclustering more difficult. In a system that manages data which is expected to be stored for a long time (which is the case for most databases!), with possible changing applications and access patterns, logical OIDs should be used to avoid performance degradation later.

3.1.1 Physical OID

The OID is usually organized as a data structure, designed to help the ODBMS achieve good performance. For example, consider the 64-bit OID used by the Objectivity/DB (illustrated in Figure 3.1):

1. A logical (federated) database can be composed of several physical databases, and the first field in the OID identifies the physical database. A physical database is mapped to a file on a server, so this field identifies the server and file where the object is stored.

2. A physical database is composed of a number of containers. The container field identifies the actual container.

3. The page field identifies the page where the object is stored.

4. The slot field identifies the slot of an object on a page.

Database	Container[1]	Page	Slot
16 bits	16 bits	16 bits	16 bits

Figure 3.1: OID in Objectivity/DB [167].

An OID organized as a data structure like the one described above helps in providing efficient access to objects, but at the same time it also imposes a strict limit on the number of databases and containers in the system. Although 2^i objects can be created during the lifetime of a database if an OID size of i bits is used, the number is much smaller in practice. In real world applications, it is impossible to exploit all these fields, and it is obvious that careful design is needed to avoid problems with the maximum numbers of containers in the system. At the same time, it is also possible to get problems because of the limited number of objects that is possible to store in one container. These problems can be eliminated by increasing the OID size, but that reduces the storage efficiency.

3.1.2 Logical OID

Logical OIDs are more flexible. Objects can be relocated, and in theory, it should be possible to exploit the whole range of possible OIDs, given a certain OID size. However, in practice, the structure of logical OIDs is often similar to physical OIDs, for example the OID structure used in Versant [46]. If such a structure was not used, OIDs from the same collection and from the same database would be distributed over the range of allocated OIDs, making the OIDX very unclustered.

The number of OIDs can be very large, and if logical OIDs are used, a fast and efficient index structure is necessary. The OIDX is typically realized as a hash file or as a tree structure [62]. Most common is the use of B-trees, but other specialized structures have been proposed: One example is the *hcC-tree* [193], another example is *direct mapping* [62], where OIDs contain the physical address of the mapping information, and the mapping information is kept in a structure organized as an extensible array.

3.1.3 Combination of Physical and Logical OID

To improve performance, it is possible to use a combination of physical and logical OIDs, as is done in Shore [136]. In Shore, physical OIDs are used at the storage manager level. However, the *Value Added Server*, which is the interface to the users/application programs, can support logical OIDs by maintaining an OIDX.

3.2 Object Storage Structure

The way an object is stored, determines the update and query costs. In general, we have two primary strategies:[2]

- Direct storage model.
- Decomposed storage models.

[1] One bit is for internal use, so that only 15 bits are used for the container number.

[2] Note that authors of the papers discussing these models, do not always use a terminology consistent with previous definitions.

3.2.1 Direct Storage Model

In the direct storage model, there is no fragmentation of an object. The object, with all its attributes, will be stored as one contiguous sequence of bytes, similar to the in-memory version of an object in most programming languages. Large objects can also be stored as a contiguous sequence of bytes, but they are normally implemented with some access method to improve access efficiency.

3.2.2 Decomposed Storage Models

In the decomposed storage models [50], complex objects are decomposed so that each tuple in a database file (set of pages), only contains one of the attributes, together with a surrogate (OID in the case of an ODBMS). In one particular model, the binary storage model, attributes are stored as `(OID,value)` tuples. Providing that the tuples from objects in a certain collection are clustered together, this keeps the number of disk-reads to a minimum. This is very beneficial in applications where set queries are frequent, but if the objects are used by application programs in a persistent programming language, the objects have to be reconstructed before delivery. To reconstruct a set of objects, join operations are needed. Several studies have been done to study these tradeoffs [11, 205].

3.3 Object Clustering

In general, an object page contains more than one object. The performance of an ODBMS depends heavily on the number of object pages it has to read and write. In order to keep this number as low as possible, we try to store objects that are expected to be accessed together, on the same page. This process is called *object clustering*, and is done by using one or more of the following strategies:

- Clustering hints.
- Cluster trees.
- Dynamic clustering.

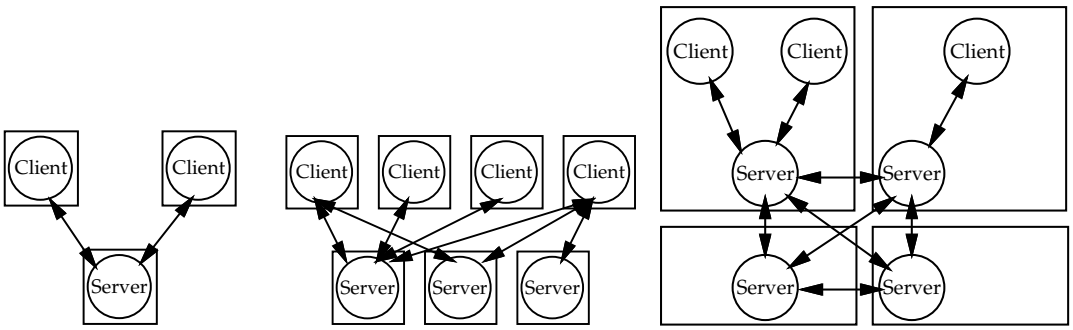
In most systems, objects that have been made persistent by a given clustering strategy remain where they are, even if the clustering policy changes (modification of the cluster tree in the case of a cluster trees strategy, or changing access pattern in the case of dynamic clustering).

3.3.1 Clustering Hints

When using clustering hints, the *application programmer* has to specify an existing object which the new created object should be stored close to (if possible). The performance of this approach is heavily dependent of an application programmer's predictions of future access patterns, and is likely to break down in more complex multiuser systems. Systems where this strategy is supported, includes ObjectStore, Objectivity and O_2.

3.3.2 Cluster Trees

Cluster trees is a more general approach to obtain good clustering. In this case, the *database administrator* specifies rules for object clustering. Typical examples of clustering strategies are to store together objects and related subobjects that are expected to be accessed together later, and members of a set that are later going to be accessed in scan operations. This strategy is supported by O_2 [165].



(a) Single server/multiple client.

(b) Multiple server/multiple client.

(c) Multiple server/multiple client (peer-to-peer).

Figure 3.2: Client/server architectures.

3.3.3 Dynamic Clustering

If dynamic clustering is supported, the *ODBMS* takes all responsibility for the clustering, and uses sampling of previous access patterns to decide where to store the objects. Algorithms and strategies for dynamic clustering strategies include the Cactis algorithms [59] and stochastic clustering [207]. A combination of cluster tree and dynamic clustering is also possible, as described by Benzaken et al. [10]. We do not know of any commercial ODBMS that supports a dynamic clustering strategy.

The performance of some object clustering techniques, under different workloads, have been studied by Tsangaris and Naughton [208]. Of those studied, stochastic clustering [207] had the best average performance.

3.3.4 Reclustering

Although not yet supported by any of the commercial systems, adaptive on-line reclustering is possible. One approach is described by McIver and King [139], and the cost of monitoring and reorganization has been studied by Gerlhof et al. [73].

3.4 Client/Server Architectures

The architecture of an ODBMS is usually a client/server variant. A client requests data, performs some operations on the data, and sends updated data back to the server. The exact division of the work between a client and its server, for example which of them is indexing data, varies between systems.

Several client/server architectures are possible. Figure 3.2 illustrates the most typical client/server architectures, with processes drawn as circles, computer nodes as squares, and communication channels illustrated with arrows:

- Figure 3.2a is a *single server/multiple client* architecture. This is the traditional and least complex architecture, supported by most systems. The clients can run on the same node as the server if desired.

- Figure 3.2b is a *multiple client/multiple server* architecture. The client itself has the responsibility of connecting to the servers containing the data, and to manage distributed commit (2-phase commit). This architecture is also supported by most commercial systems. A client can also in this case run on the same node as one of the servers or as other clients.
- Figure 3.2c is another variant of a *multiple client/multiple server* architecture, similar to the architecture of Shore ODBMS as presented in [39].[3] In this case, a client connects to *only one* server, running on the same node. The server has the responsibility of fetching remote objects/pages. A possible variant of this option, is clients residing on other nodes than the servers they are connected to. The main point is that one client only connects to *one* server, and this server communicates with the other servers as needed on behalf of the client.

3.5 Method Execution

The client and the server are in general different processes, usually executing on different nodes. When an object method is to be executed, this can be done either by the *client*, by the *server*, or by *a separate process on behalf of the client* on the server node:

1. Client node/client process: The method executes in the client's address space.
2. Server node/client process: A process is executing on behalf of the client on the server node.
3. Server node/server process. The method executes in the server's address space.

The problem with the first approach, is that all data have to be sent from the server to the client, something that easily makes the network a bottleneck. The two other approaches are (partial) solutions to this problem, but at the same time they create some new problems, which we will discuss in the following sections. Not all executed methods need to be executed in the same way. In systems that support more than one of the options above, it is possible to choose one of the options, as a way of tuning the performance.

3.5.1 Client Node/Client Process

Executing methods at the client, on the client node, is the most common in ODBMSs, and is supported by all commercial systems [8].

In applications where good page clustering has been achieved, and with only moderate data volumes, this approach works well. However, in the case of queries involving filtering operations (for example attribute selection), this approach wastes valuable network bandwidth. If filtering could be done on the server, less of the data actually has to be transported.

3.5.2 Server Node/Client Process

It is possible to run the whole client at the server node, but this can make the server node overloaded, and we do not benefit from the processing power of the client node. To solve this problem, and still avoid the drawback of the client node/client process, it is possible to execute some of the methods (or some of the query) on the server node (but note that they run as separate processes, i.e., not in the same address space as the server process itself). This approach is supported by ITASCA [8, 97].

[3] This architecture was to our knowledge never implemented in Shore.

If the client and the server run on the same node, we have more options on how to do interprocess communication. We can use message passing, as is done in general client/server communication. One step further, is to make the server buffer itself available to the client. In this case, *all clients access the same buffer*, and the design of such a buffer has to be done very carefully:

- If using a shared read/write buffer, clients as well as the server itself can write to the buffer. A shared buffer eliminates the need for a separate client buffer. A shared read/write buffer gives good performance, and locking can be done efficiently, but it gives integrity and security problems. A method can damage contents in the buffer in the case of failure, and there is no check of access permissions. We consider this approach too vulnerable without a safe interface language.

- If using a read-only buffer, clients can read from the buffer, but not write. The clients need separate buffers for modified objects/pages, or alternatively send modified objects/pages back to the server immediately (which usually will prove to be inefficient). From a performance point of view, a read-only buffer will in many cases be sufficient, because many of the typical heavy queries are read-only queries. Security is still a problem, although encryption of shared memory is possible. However, the cost of encryption would probably be unacceptable. Integrity can also be a problem if a client read data that is being updated by another client without adhering to the locking protocol.

3.5.3 Server Node/Server Process

If the server knows the contents and structure of the objects stored on the pages, it is possible to execute methods inside the server process. Systems that support this approach include ITASCA, MATISSE, POET, Objectivity, and Versant [8].[4]

Methods written in C and C++, which are popular programming languages for ODBMS applications, can literally do whatever they want, causing data integrity problems as well as damaging the server process itself. This means that special care has to be taken if general methods should be allowed to be executed by the server process. Several solutions to this problem exist:

1. Use a type-safe language as data manipulation language. This makes it easier to guarantee that the methods executed in the language can not modify privileged data in the DBMS. This approach has been used by Liskov et al. in Thor [128, 129, 130]. Variants of this approach is to use safe “data access languages”. language

2. Software-based fault isolation. In this approach, code and data are loaded into their own fault domain, a logically separate portion of the server address space, and the object code of the method to be executed is modified to prevent it from writing or jumping to an address outside its fault domain [211].

3. Interpreting the code. As Java has gained popularity, this option has become more commercially popular. Most commercial ODBMSs already support a Java binding, but in most cases, client methods are still only executed by the client. Interpreting the code is also done in Jasmine [95], where a reduced functionality C interpreter is used.

[4] MATISSE and Objectivity only support SQL queries at the server, while Versant can only execute registered events (change notifications/triggers).

4. Trusted methods. In an ODBMS, user supplied methods can be declared as trusted by the database administrator. They can then be executed in the server's address space, while untrusted methods are still executed in a separate address space. A variant of this approach is the possibility of user written "subservers" compiled into the DBMS. This is similar to the *Value Added Server* concept in Shore [39], and *DataBlades/Cartridges* in commercial ORDBMSs.

In the case of applications based on C and C++ language bindings, the trusted method approach and the software-based fault isolation are the the only realistic alternatives. However, the increasing popularity of Java makes C/C++ less popular as ODBMS application languages, and it is likely that the type-safe language alternative (using Java) will be the most popular in the future.

3.6 Data Granularity

The issue of data granularity arises in several contexts in ODBMSs. From an application program or query language point of view, data is usually accessed at object granularity. However, at the data storage level, most ODBMSs handle data at page granularity, which means that fixed size pages are read from and written to data volumes. We will now study the data granularity issues in ODBMSs, in three different contexts:

- Client-server data transfer.
- Buffer management.
- Concurrency control.

An issue not discussed here, is *page size*. The aspects of page size have been mostly ignored in ODBMS related research publications. Obviously, page size can affect performance, and among commercial ODBMSS, we see that the page sizes differ. For example, Objectivity can use different page sizes, up to 64 KB, while Versant has a fixed page size of 16 KB.

3.6.1 Data Transfer Granularity

Most ODBMSs are variants of data shipping object or page servers. Object servers have objects as the unit of transfer between the server and client, while page servers have pages as the unit of transfer.

The advantages of an object server are:[5]

- If the objects on the object pages are not well clustered, shipping the whole page is a waste of communication bandwidth.
- Understanding the concept of an object makes it possible for the server to apply methods on the object. This is very important in order to be able to do filtering operations in object-relational queries.
- Fine-grained (object level) concurrency control is easy to implement.

The advantages of a page server are:

[5]Parts of this summary are based on the descriptions by DeWitt et al. in [54].

- If objects on the object pages are well clustered, shipping the whole page can save many object requests and communication overhead for each object. This is also an issue even in the case where the client runs on the same node as the server. If requesting only one object at a time, two process context shifts are needed for each requested object (between client and server processes). This is obviously a too much, even on a relatively fast node this would limit the number of object requests per second to a number in the order of 50000.
- Fixed size pages are easier to manage than variable size objects, and space allocation for pages is easy on disk as well as in main memory.

Most commercial ODBMSs are page based. One exception is Versant, which is an object server, but with some features to avoid the performance problems related to single object accesses as described above:

1. *Get closure*, to retrieve references to all possible objects that can be navigated to, starting from a group of objects.
2. *Group read*, to retrieve a specified group of objects, for example based on the result from a *get closure* operation.

The page server architecture has, since the study of performance of alternative architectures by DeWitt et.al. [54], been considered as superior to object servers. However, that study was done under the assumption that each access to an object not resident in the client cache needed one remote procedure call, although it is noted that it would be possible for the server to simulate a clustering mechanism by figuring out what related objects might be needed. The study also appears to be misinterpreted (on purpose?) by many of the commercial companies. The paper's conclusion is actually that there is no clear winner in this study. An even more important factor, *not* considered in the paper, is that different applications often have different access patterns to the database. This means that it can be impossible to get a good clustering. Recent evaluations of real world applications, for example by Hohenstein et al. [88], support the view that object servers in many cases will perform as well as, and in many cases much better than, a page server. This is also verified by Kempe et al. [105]. One drawback of page servers that should be taken more seriously is the security and integrity risks of clients operating on pages.

In our opinion, the most important argument in favor of the object server architecture is the possibility to do some of the work at the server side. This is especially important for complex set operations, where filtering operations can significantly reduce the amount of data transfer. This has been a neglected issue in ODBMS, but we expect set operations to be given more attention in the future. This issue is also discussed in more detail in Appendix A.

3.6.2 Buffer Granularity

In the previous section we discussed the data shipping granularity. A related issue is the buffer granularity. We have the following alternatives:

- Page buffer.
- Object buffer.
- Dual buffer.

Note that buffering at the server and the clients may be handled differently. For example, the server can use a page buffer, while the clients use object buffers. However, if data transfer granularity is pages, it does not make sense to have an object buffer at the server side.

It is also important to note that multiple copies of data might reside in different client caches. Replica management is necessary to ensure cache consistency. Cache consistency is usually achieved by using pessimistic locking-based cache consistency protocols [40].

Page Buffer

Most ODBMSs use a page buffer. If objects on object pages are well clustered, a page buffer makes good use of the buffer memory. Fixed size pages are also easy to manage. Space allocation is easy, and we have no memory fragmentation problem.

Object Buffer

With an object buffer, objects are stored as independent objects in main memory, and not in the pages. This approach is beneficial if objects on the object pages are not well clustered. In that case, storing the whole page in main memory is a waste of space, because many objects in memory are not really needed there. The result will be a lower buffer hit rate than necessary when accessing objects. Another advantage is that it is possible to store objects larger than one page as one contiguous object, which is beneficial if server side execution of methods is possible.

Disadvantages of using an object buffer is that the per object overhead in an object buffer can be quite high, and we must expect some degree of memory fragmentation as well. Updates are also more complicated if we employ in-place updates. In that case, when a dirty object is to be written back to disk, it is necessary to first do an installation read of the page where the object should be stored.

Dual Buffer

A third alternative is a combination of page and object buffers. In this case, we try to keep well clustered pages in a page buffer, and objects from less clustered pages in an object buffer. This approach is used in several commercial systems, including Itasca, Ontos and Versant [52].

A thorough study of client side dual buffering by Kemper and Kossmann [108] showed that dual buffering can give a substantially higher buffer performance than a page buffer. However, the use of a dual buffer introduces several new options that makes tuning more complicated, for example when to copy an object from the page buffer to the object buffer, and when to copy a dirty object in the object buffer back to its home page. This makes it less clear how well dual buffering would perform in systems with complex workloads. Also, the study showed that dual buffering was mainly beneficial with read queries, with update queries the gain was less or negative.

3.6.3 Concurrency Control Granularity

Concurrency control can also be done at different granularities. This is usually adaptive, and can be fine grained, e.g., object, or coarse grained, e.g., page or file granularity.

3.7 Buffer Management

Keeping the most frequently used data in main-memory buffers reduces the number of disk accesses. Efficient buffer management is crucial to achieve good performance, and in this section we will discuss

buffer allocation and replacement.

3.7.1 Buffer Allocation Algorithms

With fixed size granules, for example pages, buffer allocation and deallocation is straightforward, and we have no memory fragmentation.

With variable sized granules, we will in practice have some degree of memory fragmentation. The amount of fragmentation is dependent of the amount of CPU we are willing to use to reduce the fragmentation. Using buddy allocation, which has a low CPU cost, gives a memory utilization of approximately 80%. However, it has been shown that it is easy to get a memory utilization above 90% by only a marginal increase in the CPU cost [70, 104].

3.7.2 Buffer Replacement Algorithms

The main-memory buffers can usually only keep a selected subset of the contents that are stored on secondary and tertiary storage. When an item is brought into main memory, another item has to be removed from the buffer to make space for the new item.

Buffer replacement is often LRU based. In the case of a page buffer, the pages are usually linked in an LRU chain. The overhead of an LRU chain is acceptable when the size of the pages is much larger than the extra data structures needed for the LRU chain. With a finer granularity, there is a larger number of granules, and a higher number of accesses to each of them. In this case, the traditional LRU chain can be a bottleneck:

- The memory overhead may be too high. For the LRU chain, two pointers are needed for each item.
- The CPU overhead can be too high, because we have to update the chain on every access.
- When the buffer is shared between several threads or processes, the pointers need to be protected by semaphores, and the head of the chain will often become a semaphore bottleneck.

Good approximation to LRU, useful for finer granules, are the *clock* and *enhanced clock* algorithms [61], also called second-chance algorithms. With the clock algorithm, only one overhead bit is needed for each granule, an *access bit*. For the enhanced clock algorithm, two bits are used, an *access* and a *dirty* bit.

When using the clock algorithm, the access bit is set each time an item is accessed. The buffer is treated like a circular queue. We have a *clock arm* (a pointer) that points to an item. When we need a candidate to discard during replacement, we move the clock arm clockwise until we find an entry where the access bit is not set. When we move the clock arm over items with the access bit set, we reset the access bits while we move the arm. In this way, an item will be discarded the next time the clock arm points at it, if it has not been accessed in the meantime.

With the enhanced clock algorithm, we also consider the dirty bit when deciding which item to discard. In general, it is cheaper to discard an item that is both clean and has not been accessed for a while, because it does not have to be written back before it is removed.

The advantages of using a clock algorithm are:

1. Lower cost when accessing an item, only the access bit has to be updated.

2. Less synchronization overhead is necessary. For example, no locks need to be acquired when an entry is accessed. That would be necessary when moving entries after an access in an LRU list.

3. If the access bits for the entries are stored in a packed format, i.e., access bits for several entries are stored in one machine word, the space overhead is reduced considerably. In this case, locking the word where an actual access bit resides, is necessary to get a serialized behavior,[6] in order to avoid loosing a *set bit* operation if two threads try to update different bits in a word by doing a *read word, set bit, write word* sequence. However, loosing an occasional access bit update should not seriously affect the buffer hit performance, so in practice, locking is not necessary!

3.8 Indexing

Indexing is a well-known technique used to reduce the query costs in DBMSs. In RDBMSs, only primitive attributes are indexed, but the increased expressiveness of the ODBMS data model makes new indexing techniques possible as well. There are also some aspects that is different in RDBMS and ODBMS indexing, and should be kept in mind:

- In RDBMSs indexing is not an integral feature, although very frequently employed. In an ODBMS, on the other hand, indexing is always employed. As discussed in Section 3.1, every object has an unique OID which can be used as a handle to retrieve the object.[7]

- In RDBMSs the relation as an extent, i.e., all members of the relation, is always maintained. In the case of object classes in ODBMSs, this is optional. In some systems, the extent is always implicitly maintained, while in other systems, this has to be done explicitly, with additional cost as a result.

- Although indexing primitive attributes in ODBMSs can be similar to indexing attributes in RDBMSs, the indexing is more complex due to existence of class hierarchies [112].

In the rest of this section we give a brief overview of path indexing and function materialization, which are not issues in RDBMSs, but can be important in order to achieve good query performance in ODBMSs.

3.8.1 Path Indexes

Most ODBMS query languages allow queries on path expressions (usually expressed by the *dot* notation). Several techniques for indexes supporting path expressions have been proposed. These include different *path indexes* [13, 14] as well as *access support relations* [109].

Path expressions is actually a kind of implicit join. If no path index exists, it can be cheaper to use explicit join techniques (pointer-based joins) in set queries, instead of doing pointer traversals [188].

Related to path indexing, is *field replication* [187], where the field (attribute) at the end of a path expression is replicated, and stored inside the first object in the path.

[6] If a *set bit* operation exists, this is not necessary. However, single bit operations is not always available, usually they are provided only by CISC processors, for example the Intel x86 family.

[7] In the case of physical OIDs, the indexing is implicit.

3.8.2 Function Materialization

Predicates in ODBMS queries can involve methods as well as attributes. It is possible to use precomputed values for methods to increase query performance. This technique is called *function materialization* [106].

3.9 Swizzling

Pointer swizzling is the process of converting pointers in an object from disk format (physical or logical OIDs), to memory addresses, so that subsequent object navigation operations do not have to go through an index or "resident object table" in order to find the actual object. Although swizzling has not previously been considered as a server issue, it is likely that it can increase the performance if methods can be executed by the server.

The possible gain from swizzling does not come for free. If an object has been modified, all swizzled pointers to this object have to be changed back to disk format before the object is removed from the buffer. Thus, swizzling is only beneficial if the swizzled object is referenced several times, and the update rate is sufficiently low. Several swizzling strategies exists [141, 213]. Which strategy to use, and whether to swizzle at all, depends heavily on the access pattern, and adaptable swizzling strategies might be a good alternative [107].

3.10 Query Processing

Query processing in an ODBMS can be done in much the same way as it is done in a RDBMS [110, 219]. The user[8] submits a query to the system, usually in some declarative language. This query is optimized, normalized, and transformed to some object algebra expression. After type checking, algebra optimization is performed, and an execution plan from this optimized algebra expression is generated and executed. Similar to a RDBMS, the difference in execution time between a query with good optimization and a query with bad optimization, can be several orders of magnitude.

Even though the basic techniques are the same as in RDBMSs, ODBMS query processing has many aspects which makes it more complex than query processing in RDBMSs. The most important differences are [111, 219]:

- ODBMSs have a much richer type system than RDBMSs, which only have the single aggregate type *relation*. In ODBMSs, queries can be performed on various kinds of collections, where members can be of different types.
- Encapsulation and methods: how much should the system know about the implementation of a method, and should it be able to break encapsulation?
- An object may reference other objects, and accessing these objects involves path expressions/implicit joins.
- In ODBMSs, indexing can also be done on access paths, not only on primitive attributes, as in RDBMSs. Class hierarchies also complicate the use of indexing.

[8] User in this context can be either a person giving a command to the DBMS, or an application program sending a request to the DBMS.

- Inheritance can make it difficult to determine the access scope of a query. This makes efficient object access more complicated.
- Cyclic queries need special attention.

These differences, the availability of different kind of indexes, and the choice between forward and reverse traversal (whether to start on the target class/root of query graph, or at any intermediary) increase the number of possible query plans. This makes the process of evaluating query plans more costly and difficult in an ODBMS compared to a RDBMS.

3.11 Parallel ODBMSs

The performance of a DBMS can be increased by increasing the available hardware resources. This means more powerful hardware, or duplication of resources. Employing more powerful hardware is one solution that has been considered "easy", as it has no consequences for the implementation of the ODBMS itself. However, this strategy is only cost effective up to a certain point. After that, duplication of resources, i.e., a larger number of CPUs and disks, is needed. It should also be noted that this strategy is more difficult than it looks, because the CPU speed, memory- and disk bandwidth have to be kept in balance.

If using more than one CPU and more than one disk, work has to be distributed over the CPUs and the disks in a way that make all of them busy most of the time, and avoids any single bottleneck. Even though parallelization of "simple" set queries are well understood from the work on parallel query processing in RDBMSs, parallel query processing in ODBMSs is less mature. There are several reasons for this, but the most important is that ODBMS query processing can be very complex in itself. The fact that the architectures of most systems are based on data shipping, makes filtering on the servers difficult, and it is difficult to keep the data transfer volume at a moderate level.

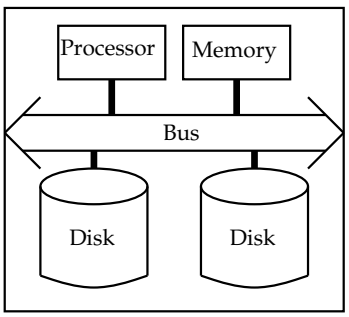
In this section, we discuss parallelization in ODBMSs. We start with a presentation of alternative parallel architectures, and then give an overview over issues in parallel query processing. To set the work presented later in this thesis in context, we also summarize work on previous parallel ODBMSs.

3.11.1 Alternative Parallel Architectures

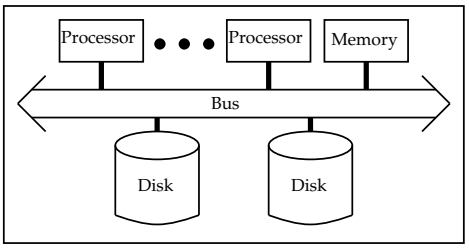
Even on single processor computers, as illustrated in Figure 3.3a, multiple disks are common, and if used to host a DBMS, are necessary in order to provide redundancy in the case of media failure. A larger number of disks can also be used to improve the data transfer bandwidth and transaction throughput, for example by using RAID technology. The advantage of this approach, is that it is relatively easy to utilize the disks.

Current servers are often symmetric multiprocessors (SMP), with a number of disks attached. In an SMP, all processors have equal access to memory and disks. This is called a *shared everything* configuration (see Figure 3.3b). The limiting resource in an SMP node is the bus, which soon gets saturated as more processors are added. The advantage with this approach, is that it is relatively easy to utilize the CPUs if the DBMS is implemented as a multithreaded or multiprocess server.

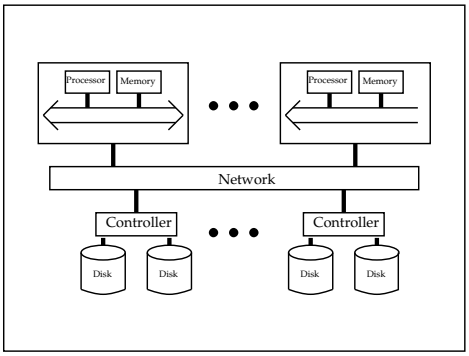
A further improvement is *shared disk*, where processors have equal access to the disk system, but not on the same bus (see Figure 3.3c). In a shared disk configuration, issues such as fragmentation and clustering are easier than for a shared nothing approach. Shared disk is the traditional mainframe approach, and has not been very common in the case of systems made from off-the-shelf hardware.



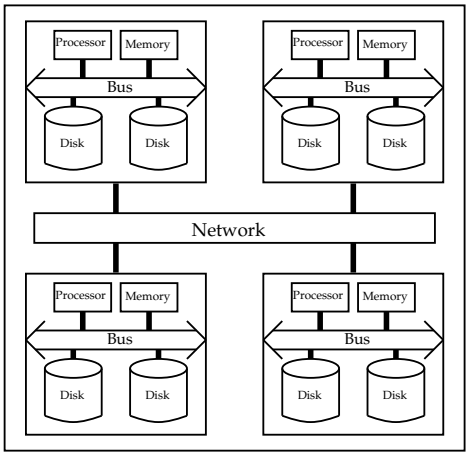
(a) Single processor computer.



(b) Shared everything configuration.



(c) Shared disk configuration.



(d) Shared nothing configuration.

Figure 3.3: Alternative parallel architectures.

However, with the increasing popularity of storage area networks, using Fibre Channel, we expect it to be more common in future systems.

The most scalable approach is *shared nothing* (see Figure 3.3d). In this case, we have a number of nodes which each has local memory and a number of disks. The nodes communicate through some kind of interconnection network, using message passing. A shared nothing computer is also commonly called a *multicomputer*, but nowadays, a cluster of workstations connected to high bandwidth network is also suitable as a platform for a parallel database server.

3.11.2 Parallel Query Processing

We will in this section concentrate on issues related to shared nothing servers. However, it should be noted that some of the work done in the context of multiprocessors is also relevant, including the research on parallel query evaluation done by Härder et al. [90], and on optimizing and parallelizing ODBMS programming languages by Lieuwen et al. [127].

Data Distribution

Optimal allocation and fragmentation is very important, but complex objects, object classes and inheritance increase the size of the solution space for the data distribution problem. These issues have been studied by a number of researchers, including Gruber and Valduriez [82], Karlapalem et al. [103], and Ghandeharizadeh et al. [76]. Ghandeharizadeh et al. also show how replication can be efficiently employed to increase performance. It has also been shown that in a parallel system where it is possible to store most of the working set in main memory, utilizing the aggregate memory of all the nodes can significantly improve the performance [209, 210]. A more detailed discussion if the data distribution problem is given in Chapter 11.

Query Processing

Optimal data distribution is in general heavily linked to queries on the data. This has been studied in detail by Kim [111] and Chen and Su [48]. In many cases, redistribution of data can be efficient [121]. Parallel join algorithms for set-valued attributes is described by Lieuwen et al. [126].

3.11.3 A Brief Overview of Parallel ODBMSs

We will now give a short description of previous parallel ODBMSs. With one exception (Objectivity), all the described systems are research prototypes. Several of the prototypes (ADAMS, AGNA, and PPOST) are systems built as a part of a PhD work, and seems to have been abandoned after the PhD work was finished. One of the systems, Shore, has never been implemented as intended (multi-server version), while the work on Multicomputer Texas seems to have been restricted to the cited paper only. All in all, these systems illustrate well how immature the area of parallel ODBMSs is. Although the complexity of the area is one reason why so little research has been done, it is likely that the advent of ORDBMSs, which had a negative impact on the amount of ODBMS research in general, also has been important. However, the results from some of these projects have been convincing enough to make us believe that this is an area that deserves more active research.

In addition to the systems described here, some of the commercial systems also provide some support for multiple servers. However, the application programmer has the responsibility for the distribution of data, and the support for distribution mostly means “the system supports 2-phase commit”.

ADAMS

ADAMS is a parallel "data management system", running on a network of workstations [86, 174]. It has many ODBMS features, but lacks concurrency control and recovery, which are important features of a database system. Its main application area is SSDBs, an area where these features often are of minor importance.

ADAMS employs the decomposed storage model for object storage, and declusters objects by the OID. The system processes set operations by streaming of data like most parallel RDBMSs, and has shown good performance and scalability.

AGNA

AGNA [145], a persistent programming system, is based on a LISP like environment. The system is designed to run on a shared nothing multicomputer. Objects are referenced by their heap address. The heap is global, and distributed over the nodes in the system.

Bubba

Bubba is a highly parallel DBMS [28]. Its application area is data-intensive applications. Data is horizontally partitioned (which favors objects with few references to other objects), and performance depends on executing operations at the node where the object resides. This is supported by the use of automatic parallelization by the Bubba compiler and an analytical model for data placement.

Eos

Eos, which is short for *Environment for building Object-based Systems*, is a distributed single-level store [81]. Distribution of data over the nodes is supported by facilities in the Mach operating system. Eos is supposed to be scalable, but there are no data on performance that can support this claim.

Multicomputer Texas

The Multicomputer Texas [18] is a parallel object store based on the Texas object store [189]. Multicomputer Texas has been implemented on a Fujitsu AP1000 multicomputer and a network of workstations. A modified Texas object store is run on each node, providing a global persistent address space. In this way, we can see it as a distributed shared memory implementation. No support for parallel query processing or efficient declustering of data is provided, and performance is highly dependent of the locality of data to be accessed at the nodes. In this respect, we feel that the practical value of the implementation is limited.

Objectivity

Objectivity [167] is the only commercial ODBMSs that is able to use parallelism to significantly increase performance. Objectivity is a page server ODBMS, employing NFS (Network File System). The servers run on ordinary network connected workstations, and the distribution of data can be used to increase performance as well as availability (by replication). Objectivity has been chosen as the ODBMS to be used in the CERN RD45 project, where experiments will generate an amount of 1 PB of data a year, and up to 1.6 GB/second data rate [44, 45, 46, 47].

OSAM*.KBMS/P

OSAM*.KBMS/P is a parallel, active, object-oriented knowledge base server [201]. The server runs on a shared nothing computer (nCUBE2), and the clients on workstations connected to the nCUBE via Ethernet. The knowledge base is partitioned class-wise, i.e., all members of a class is stored on the same node. A global transaction server is used to supervise executions.

PPOST

PPOST [21, 22, 23] is a parallel, main-memory object store, implemented on a cluster of workstations. Because transactions are committed to disk sequentially, the architecture is only suitable for application areas with a small number of concurrent transactions, and where transactions are short in time, but with high data bandwidth.

Shore

Shore [39], Scalable Heterogeneous Object REpository, is a persistent object system with many novel features. The most interesting in the context of this section, is the introduction of a symmetric peer-to-peer server architecture. All application programs in the system are connected to *one server*, running on the same node. This server is the gateway to the DBMS. Unfortunately, multi-server Shore has never been implemented, although some research have been done on parallel set processing by the use of ParSets [57], and global memory management [209, 210].

The storage manager of Shore has later been used in Paradise [55, 173], a parallel DBMS for GIS applications. Paradise is described by the authors as object-relational, rather than object based (or object-oriented).

3.12 Summary

The design of an ODBMS introduces new issues not found in RDBMSs, and we have in this chapter discussed design issues in ODBMSs, with an emphasis on issues that are specific for ODBMSs. The discussions in this chapter will be used as a background for the description of Vagabond, as well as establishing the terminology which will be used in the rest of this thesis.

Chapter 4

Temporal Database Systems

The rest of this thesis will concentrate on the design of a temporal ODBMS. It is therefore appropriate to start with an introduction to temporal DBMS in general, the terminology, and related work in the area. This chapter is not intended as a complete study on temporal databases, its purpose is only to give the reader the necessary background to comprehend the rest of this thesis.

4.1 What is a Temporal DBMS?

A temporal DBMS is a DBMS that supports some aspect of time. Informally, this means that data is associated with time, and that a tuple (temporal RDBMS) or object (temporal ODBMS) can exist in several versions, each version being valid in a certain time interval. An example is the salary of a person. Each time the salary is changed, a new version of the person's salary tuple/object is created. In a temporal DBMS, this versioning, related to time, is supported and maintained by the system, which also provides support for querying the data.

Even before people started to think about temporal DBMS as an area of its own, time has been related to data in a database, for example by the use of an attribute containing a time value. Typical examples are attributes such as "birth day" and "hiring date." However, there has not been support for temporal aspects in the query languages, and queries and management have been done in various ad-hoc ways. In our terminology, we call this uninterpreted attribute domain of date and time *user-defined time*. A temporal DBMS is now defined as a DBMS that supports some aspect of time, *not counting user-defined time*. A traditional, non-temporal DBMS is called a snapshot database system. Not all data stored in a temporal DBMS needs to be temporal and the data that is not temporal is called *snapshot data*.

Even though temporal databases have a long history, it is only very recently that research in this area really has taken off, and more importantly, the industry has begun to signal interest in the work. Two projects have in particular contributed to the current interest and results in the area: the *consensus glossary of temporal database concepts* [101], and the *temporal structured query language* (TSQL2) specification [191]. The consensus glossary is recommended by a significant part of the temporal database community, and the definitions and terminology in this chapter are based on that glossary.

4.2 Data Models

Many temporal data models have emerged during the years. A presentation of these is outside the scope of this chapter, and we will only give a brief overview of the most important aspects of these

models.

4.2.1 The Time Domain

Time models can be linear, branching, or cyclic. In a linear time model, time advances from the past to the future in an ordered step by step fashion. In a branching time model, time can split into several time lines, each representing possible event sequences. In a cyclic time model, we can also have recurrence. One example is a week, where each day recurs every week [218].

The time line itself can be either discrete or continuous. If discrete, each point in time has a single successor, like natural numbers. If continuous, there are no gaps, similar to real numbers. In most models, a discrete time line is used, where we have a non-decomposable time interval of some fixed, minimal duration of time called a *chronon*. Important special types of chronons include valid-time, transaction-time, and bitemporal chronons. A data model will typically leave the particular chronon duration unspecified, to be fixed later by the individual applications, within the restrictions posed by the implementation of the data model.

4.2.2 Aspects of Time

The most common aspects of time in temporal DBMSs is transaction time and valid time.

Transaction time: A database fact is stored in a database at some point in time, and after it is stored, it is *current* until logically deleted. The transaction time of a database fact is the time when the fact is current in the database and may be retrieved. Transaction times are consistent with the serialization order of the transactions. Transaction-time values cannot be later than the current transaction time. Also, as it is impossible to change the past, transaction times cannot be changed. Transaction times may be implemented using transaction commit times, and are system-generated and -supplied. It is important to note that each update of an object creates a new current version. We call the non-current versions *historical versions*.

Valid Time: The valid time of a fact is the time when the fact is true in the modeled reality. A fact may have associated any number of instants and time intervals, with single instants and intervals being important special cases. Valid times are usually supplied by the user.

Valid times can be open-ended intervals. One example of this, is the existence of a house. We know when it was built, but now when it will be removed.

Bitemporal: Temporal DBMSs can also support a combination of these aspects, bitemporal data. Bitemporal data have exactly one system supported valid time interval, and exactly one system-supported transaction time.

A bitemporal interval is a region, with sides parallel to the axes in a two-space of valid time and transaction time. When a bitemporal interval is associated in the database with some fact, it identifies when that fact was true in reality (during the specified interval of valid time), and when it was logically in the database (during the specified interval of transaction time).

A good example to illustrate the use of transaction and valid time, is a GIS database. In this database, objects such as houses and roads are stored. When an object is stored in the database, it is timestamped with the transaction time. However, the time this actual object existed in the real world is

in general different from the time it was entered into the DBMS. For this purpose, it is also necessary to store the valid time of the objects in the database.

4.3 Temporal Queries and Query Languages

Several query models for temporal databases have been proposed, and others are likely to be proposed in the future. In practice, most research on temporal databases is now based on *TSQL2* [191], an extension of SQL-92. TSQL2 provides language constructs for schema definition, schema evolution and versioning, and querying and updating temporal relations. The goal of the language design was to form a common core for future research, more than designing a language for the commercial market, but work is currently under way to incorporate TSQL2 into SQL3 [190].

TSQL2 employs a simple data model, based on the relational data model. In the conceptual model, the *bitemporal conceptual data model*, tuples are timestamped with a bitemporal interval.

Queries are performed on a collection of tuples. In addition to the traditional relational operators, temporal operations are also needed. We will now present the most important temporal operations, with examples based on TSQL2 [191].

4.3.1 Temporal Selection

With temporal selection, it is possible to retrieve data valid at a certain time, *valid time selection*, or current at a certain time, *transaction-time selection*. It is also possible to do a selection based on both valid and transaction time, *bitemporal selection*.

Several new operators are included in TSQL2 to be used in the temporal selection predicates, including operators for comparison of timestamps:

- **FIRST(event, event)**
- **element PRECEDES element**
- **period CONTAINS period**

To illustrate temporal selection, consider an example query from an employee database that lists all of the employees who worked during all of 1991:

```
SELECT Name
FROM Employee
WHERE VALID(Employee) CONTAINS
        PERIOD(DATE '01/01/1991', DATE '12/31/1991')
```

4.3.2 Temporal Projection

In a query or update statement, temporal projection pairs the computed facts with their associated timestamps,[1] usually derived from the associated timestamps of the underlying facts. The generic notion of temporal projection may be applied to various specific time dimensions. For example, valid-time projection associates with derived facts the times at which they are valid, usually based on the valid times of the underlying facts.

[1] Note that a timestamp can also be an interval.

4.3.3 Temporal Join

A temporal natural join is a binary operator that generalizes the snapshot natural join to incorporate one or more time dimensions. Tuples in a temporal natural join are merged if their explicit join attribute values match, and they are temporally coincident in the given time dimensions. As in the snapshot natural join, the relation schema resulting from a temporal natural join is the union of the explicit attribute values present in both operand schemas, along with one or more timestamps. The value of a result timestamp is the temporal intersection of the input timestamps, that is, the instants contained in both.

4.3.4 Coalescing

Associated with each tuple in a temporal relation is a timestamp, denoting some period of time. In a temporal database, information is "uncoalesced" when tuples have identical attribute values and their timestamps are either adjacent in time or share some time in common. Coalescing is similar to duplicate elimination in conventional databases, although potentially more expensive [25]. Its purpose is to effect a kind of normalization of a temporal relation with respect to one or multiple time dimensions. This is achieved by packing as many value-equivalent tuples as possible into a single value-equivalent one.

Example: Given two tuples with the same non-temporal attributes and valid in the time intervals [40, 50> and [45, 60>, respectively.[2] The result of a coalescing these tuples is *one* tuple, with the same non-temporal attribute values as the two input tuples, and the time interval [40, 60>.

4.3.5 Temporal Aggregation and Grouping

The main difference between traditional value-based aggregation and grouping, and temporal aggregation and grouping, is the inclusion of time in the domain of aggregates, and the possibility to group on time. For example, **MIN(VALID(R))** can be used to select the value of the oldest or earliest tuple in a table. In addition to the traditional aggregate functions, new functions can be useful in temporal databases. One example is the **RISING** operator in TSQL2, which is defined to return the longest period which a numeric value was monotonically rising.

Time can be used as basis for the partitioning in the grouping part of the aggregation. The timeline can be divided into partitions, i.e., into time periods. For example, to compute the average salary for each 3 month period along with the start date of the period, the following query can be used:

```
SELECT AVG(Salary), BEGIN(VALID(E))
FROM Employee AS E
GROUP BY VALID(E) USING 3 MONTH
```

4.4 Programming Language Bindings

In non-temporal ODBMSs, ODMG's OQL or similar query languages can be used for ad-hoc queries. Similar to the way OQL is a superset of the part of standard SQL that deals with databases queries, it is possible to design a temporal OQL that is a superset of TSQL2. One such approach has been described by Fegaras and Elmasri [67]. However, one of the main advantages of ODBMSs is the avoidance of

[2] $[T_1, T_2>$ is short for the time interval from T_1 to T_2, including T_1 but not T_2 (open-ended upper bound).

the language mismatch by providing computationally complete data manipulation languages with no mismatch between language and storage. In the ODMG standard, language bindings based on C++, Java and Smalltalk are described. Such language bindings are also needed for temporal ODBMSs. It should also be noted that in order to use methods in queries, these issues have to be resolved.

A general purpose programming language is only designed for current data. Integrating support for access of historical data into a programming language introduces a lot of interesting but difficult issues, including:

- Which object interface/signature to use when accessing a historical object version. The schema might have been changed since the historical version was created, so that the current interface to the class is different from the one previously used.
- Which method implementation to use when calling methods in historical objects. One straightforward approach is to use the implementation that was current at the same time as the actual object version was current. However, this is not necessarily what we want, if the reason for a new implementation of a method was a bug in the previous version. This problem can be solved by providing the necessary information at schema change time.
- How to integrate *time* into the syntax of the programming language.

In the rest of this section, we will discuss the integration of access to historical data into a general-purpose programming language.

4.4.1 Temporal C++ Binding

In this section, we describe two approaches that extend the C++ language binding with support for access to historical data in a transaction-time ODBMS. The first approach is based on the language binding used in POST/C++ [202], while the second is to our knowledge new. The concepts of these approaches can also be employed for a Java language binding.

Explicit Object Version Access

The easiest way to integrate object version access into the programming language is to provide explicit access to the versions. This is the way it is done in POST/C++ [202]. Given an OID, the program can be given a pointer to a historical version valid at a particular time by calling a function `snapshot(OID,time)`. It is also possible to create iterators that can be used to navigate the versions of an object in chronological sequence.

This approach should be easy to use and understand, but if it should be possible to call a method in a historical object version that accesses other objects, the historical version must itself do the necessary operations in order to retrieve the objects valid at the same time as when the version was created.

The Explicit Snapshot Approach

A better and "cleaner" alternative than the one described above is to use explicit snapshots. Before calling a method in a historical version current at time `ts`, we set the snapshot time with a call to the function `set_snapshot(d_Timestamp ts)`. After the `set_snapshot()` function has been called, an access to a particular object will be to the object version current at time `ts`, *even though the*

reference is through a `d_Ref`.[3] A call to `set_current()` will set accesses back to normal, i.e., an access to a particular object will be to the current object version. Methods called in historical objects should in general be immutable, i.e., read-only methods. The advantage of this approach is that all object versions accessed will be object versions valid at the same time.

All access, creation, modification and deletion of persistent objects must be done within a transaction. In the ODMG C++ binding, transactions are implemented as objects of the class `d_Transaction` [43]:

```
class d_Transaction{
public:
      d_Transaction();
      ~d_Transaction();
      void begin();
      void commit();
      void abort();
      void checkpoint();
      ...
private:
      ...
};
```

The `set_snapshot(d_Timestamp ts)` and `set_current()` functions are performed in the scope of a certain transaction, so it is reasonable to extend the ordinary C++ transaction class with these methods, for example with a derived class based on `d_Transaction`, which includes these functions as methods:

```
class d_TTransaction:public d_Transaction{
public:
      void set_snapshot(d_Timestamp ts);
      void set_current();

private:
      ...
};
```

Each temporal object can be viewed as a collection of object versions. A collection interface should exist to make it possible to iterate through the object versions in a flexible way. This collection interface is also used when assigning a value to a `d_HRef` variable (a reference to a particular version), i.e., assigning an object version to the `d_HRef`.

4.4.2 To Bind or not to Bind?

We have now outlined how objects could be accessed through a standard language binding. It should be noted that the problems involved in this integration also can be an argument *against* doing this. It is possible that only allowing access to historical versions through a temporal query language is less error prone and more efficient than providing access through an explicit language binding. A more in-depth study of the language binding, and whether to have it at all, is interesting further work.

[3] A `d_Ref` is a reference to an object.

4.5 Vacuuming

When an object has been deleted in a snapshot database, it can not be accessed later. Usually, the space occupied by the object will be overwritten by new data after it has been deleted. Temporal databases, however, follow a non-deletion strategy, where logically deleted data are kept in the database. Even though storage cost is decreasing, storing an ever growing database can still be too costly in many application areas. A large database can also slow down the speed of the database system by increasing the height of index trees (even though this can be avoided with multi-level indexes, at the cost of a more complex system). As a consequence, it is desirable to be able to physically delete data that has been logically deleted, and delete non-current versions of data that is not deleted. This is called *vacuuming*. Note that the term *vacuuming* has also been used for the migration of historical data from secondary storage to cheaper tertiary storage. In this thesis, we will use the term for *physical deletion* only.

4.6 Implementation Issues

We will do a more detailed discussion of some implementation issues in temporal DBMS later in this thesis. In this section, we will restrict the discussion to an overview of some of the most important work in the area.

4.6.1 Partitioned Storage

Storage of data in a temporal DBMS is not very different from storage of data in a traditional DBMS. However, because current data tend to be more frequently accessed than historical data, data is often partitioned into a *current store* and a *history store*. The two stores can utilize different storage formats, and even reside on different storage media [4]. In this way, frequently accessed data is clustered together, stored on fast storage media, while historical versions can be stored on slower but cheaper storage media. The total storage cost is reduced, similar to the goal of general storage hierarchies.

4.6.2 Timestamp Representation

Timestamps can be viewed from two levels: logical and physical level. The logical level is the user's view of the values, for example from a query language. At the logical level, the timestamp may look like "Dec 4 22:14:44 1998". It can also look like "1998/12/04", in a different format, at a lower granularity. However, physically, the timestamp is usually represented differently. We have several goals we want to achieve:

1. High precision. For many applications, precision down to day or hour is enough, while other applications need finer granularity. This is especially important for transaction-time databases, where we want objects from different transactions to have unique timestamps.

2. Large range. In the case of valid time, a timestamp should ideally be capable of representing all points in time, from the Big Bang to Armageddon. However, in a transaction-time database, we can accept a smaller range, from the day the system is first used, and to "some time in the future".

3. Low storage cost. To keep storage costs down, the number of bytes used to represent a timestamp should be as small as possible, given the other constraints.

4. Low processing cost, for example when creating timestamps, comparing timestamps (including ordering of timestamps), and translating between different calendar representations.

As can be observed, high precision and large ranges conflict with low storage cost. Given a certain storage cost, high precision and large ranges are conflicting goals. Low storage cost conflicts with low processing cost, because efficient storage of a timestamp will often imply transformation before and after processing.

Alternative timestamp representations can be classified as:

1. One-field alternative, often used in operating systems. In Unix, 32 bits are used to represent the seconds since its origin. This format is very space efficient, and results in low processing cost. However, the range, 136 years, is too small for a general purpose valid time temporal database. Although the range is large enough for a transaction-time temporal database, one should keep in mind that some of the data stored in temporal databases will be used some time far in the future, so that one should consider a larger range. In many applications, the precision (seconds) is too small as well. However, both precision and range can easily be increased by increasing the number of bits in the timestamp.

2. Multi-field timestamps, as used to represent time in many commercial RDBMS. In this case, there are separate fields in the timestamp for year, month, day etc. In each field, the actual year/month/day can be stored by using packed decimals or a string representation.

In a study of this issue, Dyreson and Snodgrass [60, 191], proposed a new timestamp format to solve the problems above. In their timestamp format, special values designate special times as *now* and *forever*. They also made the observation that users have a telescoping view of time, times close to *now* should be represented with finer granularity than times further in the past or in the future. They can be represented with an extended range and coarser granularity. The proposed timestamp representation can have different lengths: 32, 64, and 96 bits.

4.6.3 Indexing Temporal Databases

To support efficient retrieval of temporal data, indexing is necessary. Much research has been done in this area, and a comprehensive survey of indexing time-evolving data has been done by Salzberg and Tsotras [178]. This issue will be discussed in more detail in Chapter 8.

4.6.4 Temporal Query Processing

Even though most other aspects of temporal databases now seems to be well explored areas, the amount of publications on temporal query processing is still relatively small. One of the reasons for this, is that much of the other work (and implementations) have used a *stratum approach*, in which a layer converts temporal query language statements into conventional statements executed by an underlying DBMS [100]. Although this approach makes the introduction of temporal support into existing DBMSs easier, we do not see it as a long-term solution, because temporal query processing with this approach can be very costly.

Previous work on temporal query processing includes the work by Leung and Muntz [122, 123], which was a study of query execution on a data stream with tuples with increasing timestamps. That work was also done in the context of multiprocessor database machines [124]. An algorithm for evaluation of valid-time natural join has been presented by Soo et al. in [192]. Optimization of

partitioning in temporal joins has been described by Zurek [220]. Other important work includes aggregation algorithms [116], a study of parallel aggregation [72], and coalescing [25].

In the context of query processing in temporal ODBMSs, we are only aware of one paper, on parallel query processing strategies for temporal ODBMS by Hyun and Su [94].

4.7 Temporal ODBMSs

The area of temporal *ODBMSs* is still immature, as is evident from the amount of research in this area, summarized in the *Temporal Database Bibliography*, last published in 1998 [216]. The main reason for this low research activity is probably the number of problem still unsolved in the less complex case of temporal RDBMSs.

Most of the work in the area of temporal ODBMSs has been done in data modeling, while less have been done on implementation issues. systems have been implemented [24]. Common for most of these, is that they have only been tested on small amounts of data, which makes the scalability of the systems questionable. In most of the application areas where temporal support is needed, the amount of data will be large, and scalability is an important issue.

In the area of temporal ODBMS, we are only aware of one prototype, POST/C++ [202]. However, the indexing technique used in POST/C++ is not scalable. Good performance is only possible as long as the OID index fits in main memory (see Section 8.3.2 for a description of the indexing in POST/C++). In addition, there are implementations of temporal object data models on top of traditional ODBMSs, for example TOM, built on top of O_2 [194].

4.8 Summary

We have in this chapter given a short introduction to the terminology and most important issues in temporal database management. For more in depth discussion of the issues, we refer to the publications cited in this chapter.

Chapter 5

Log-Only Database Management Systems

Most current database systems are based on in-place updating of data combined with write-ahead logging. In this chapter we describe the alternative log-only approach, and describe its advantages and disadvantages. We describe the page-based and object-based alternatives, and why we consider the object-based alternative as the most interesting. We finish the chapter with an overview of systems that are based on log-only or related techniques.

5.1 The Log-Only Approach

In a log-only approach, data as well as metadata are written contiguously to the log. Already written data is never modified, new versions of objects or pages are simply appended to the log. Logically, the log is an infinite length resource, but the size of the physical storage is of course not infinite. This problem is solved by dividing the physical storage into large, equal sized, physical *segments*, as illustrated in Figure 5.1. A typical segment size will be in the order of 512 KB to 1 MB. When all data residing in one segment is outdated or moved to another segment, the segment can be reused. In the description in this chapter, we will assume that the log resides on disk only, but in general, the log can also reside on tertiary storage. The log also contains *checkpoint blocks*, which are used to store checkpoint information. The checkpoint blocks are stored in fixed positions in the log.

Writes are always done sequentially, normally one segment at a time. This is done by writing data and index nodes, possibly from many transactions, in one write operation. The segment size is a tradeoff between different, partly conflicting, goals: to improve write efficiency, it is desirable that the segments written are as large as possible. On the other hand, large segments can make response time longer, because writing large segments will block for read operations, and we have to wait for more transactions during group commit. Smaller segments reduce the blocking time for waiting read operations, but they also result in less efficient writing, and a larger number of segments (which means more overhead).

Because data is always written to a new place after having been updated, an index is necessary to be able to subsequently retrieve the data. This index structure is also written to the log, interleaved with the data. If the granularity of data is objects, an OID index (OIDX) is needed, and if the granularity is pages, a page index is needed (in the latter case, a page identifier is part of the OID of an object, similar to physical OIDs in traditional ODBMSs). The granularity of reading is object or pages. When a stored object or page has to be retrieved, only the desired object or page is read, not the whole

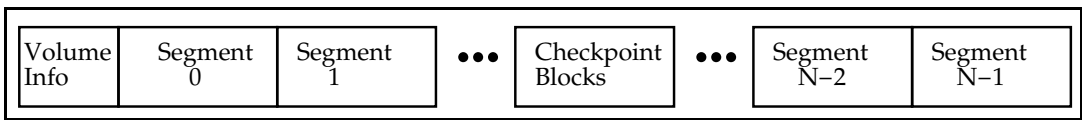


Figure 5.1: Disk volume structure.

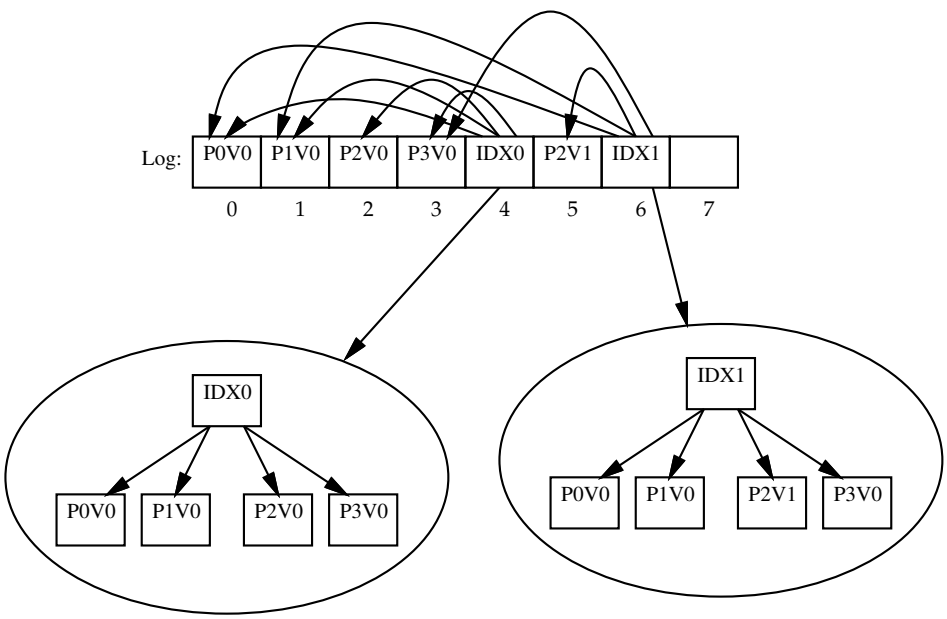


Figure 5.2: Data and index in a log-only ODBMS.

segment it is stored in.

5.1.1 Example of Log Writing

We now give an example to illustrate the log writing. In this example, the data granularity is a page, and a page index is interleaved in the log. Which page to retrieve when an object is requested is given from the OID of the object, which contains a page identifier.

Figure 5.2 illustrates how data pages and index nodes are interleaved in the log. On top of the figure is the logical log, which is a sequence of pages. Pages denoted PiVj are data pages, where i is the page number, and j is the version number or timestamp of the page. Pages denoted IDXi are index pages. The index pages will in general be part of an index tree, but to keep this example simple, we assume the number of pages is low enough to be able to store all index entries on one page.

At time t_0, a transaction allocates four pages, which are written to the log. After the transaction commits, the index node IDX0 is written, so that pages can later be accessed via this index. Later, at time t_1, a new transaction modifies page number 2 (whose first version was denoted P2V0). The new version of the page (page P2V1) and a new version of the index (index node IDX1) are written to the log.

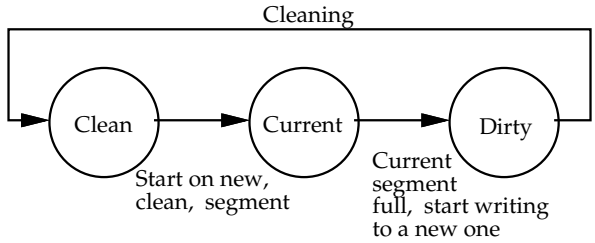


Figure 5.3: Segment states.

As can be seen from the figure, the previous versions of pages are still stored in the log, in addition to the current versions. The two versions of the database are illustrated in the figure, with arrows from the respective index nodes. Index node IDX0 indexes the database as of time t_0, and index node IDX1 indexes the database as of time t_1. As illustrated in this figure, only the current versions can be accessed from IDX1. If we want to be able to access old versions of data, we can use a multiversion index.

Note that even we in this example write data and commit sequentially, this is not necessary in practice. As will be described in more detail later in this thesis, data from different transactions and committing transactions can be interleaved.

5.1.2 Log Operations

A segment can be in one of three states, as illustrated in Figure 5.3. A segment starts in a *clean state*, i.e., it contains no data. The segment currently being written to, is called the *current* segment. When the segment is full, we start writing into a new segment. The new segment now goes from the *clean* state, to *current*. The previous segment is now *dirty*, it contains valid data (note that dirty in this context has nothing to do with main-memory state versus disk state, as the term is most frequently used). Information about the status of the segments is kept in the *segment status table* (SST), which is kept in main memory during normal operation.

If system load is low, or transactions are mostly read-only, only small amounts of new data will be created. In this case, update transactions in the commit phase, waiting for data to be written to disk, will experience long delays if we try to fill up the segments before we write. This is not acceptable, and can be solved by writing subsegments (also called partial segments). When writing subsegments, we write more than one logical segment into one physical segment. For example, if the physical segment size is 512 KB, we can instead write 4 subsegments of size 128 KB into the physical segment.

At regular times, a checkpoint operation is performed. In the checkpoint operation, we write enough information to the log to make the current position in the log a consistent starting point for recovery.

Recovery in a log-only database can be performed very fast, since there is no need to redo or undo any data. Only segments written after the last checkpoint need to be processed. At recovery time we simply do an analysis pass from the last known checkpoint to the end of the log, where the crash occurred.[1]

As data is updated or deleted, old segments can be reused. Updated an deleted data will leave

[1]However, as we shall see later, it can be beneficial to extend the amount of data that needs to be read at recovery time, to increase performance under normal operation.

behind a lot of partially filled segments, the data in these near empty segments can be collected and moved to the current segment, thus freeing up space in the old segments and making the old segments available for reuse. This process is called *cleaning*. For each segment, the SST contains a live byte counter. When data is deleted, this counter is decremented, so that we know which segments are good candidates for cleaning.

5.2 Advantages of a Log-Only Approach

Because the log-only, no-overwrite approach, is radically different from the techniques used in current systems, it is appropriate to describe the advantages of this approach.

Transparent Compression of Data. Objects are not written to the same physical location every time, and as a result, there is no need to reserve space for the maximum size of the compressed object. Even if compression ratio and the corresponding storage size change, no space is wasted.

Easier On-Line Backup. The written segments are time stamped, and with a no overwrite strategy, it is enough to know the last time of backup to know where backup should be started now. Backup could also be done on-line, and again, even if we stop backup when the load is high, we know where to continue in less busy periods.

Flash Memory. Very high performance can be achieved if we use fast non-volatile memory instead of disk. One example of such an storage technology is flash memory. Flash memory is byte readable, and fast, but write/erase has to be done blockwise. This suits a storage strategy with no in-data modifications.

Write-Once Memory. With write-once storage, for example optical disks, there is a need for a no-overwrite strategy.

RAID Technology. Disk access times and bandwidth improves at a much lower rate than main memory, and parallel disk systems are necessary to get high performance. To benefit from RAID technology, the write blocks have to be much larger than those used in traditional systems. In addition, in normal systems, sequential writes are only about 3-5 times faster than random writes, while in RAID, sequential writes can be up to 20 times faster than random writes [196]. One of the reasons for this difference is the writing of parity blocks, which is necessary in order to be able to do media recovery in the case of a disk failure.

High-Bandwidth Applications. In many supercomputing applications, and more recently also in OLAP applications, computations are done on large matrixes and arrays. To be able to do operations on these large structures, it is often necessary to break them into chunks which can be processed independently. It is necessary to retrieve and store these chunks efficiently. The same applies to storage of multimedia data, for example video. Until now, only file systems have been able to offer the desired performance. However, there is a demand for some of the services offered by database systems in these areas: access control, concurrency control, and recovery. However, performance close to file system performance is necessary for a DBMS to be applicable.

Group Commit. Group commit, in addition to giving us larger writes, also gives opportunity for more intelligent clustering of objects from different transactions.

Fast Crash Recovery. The log-only approach has similarities to shadow storage. Even though the use of shadow storage can result in performance problems, it also has a very nice and interesting feature: very fast crash recovery. By never updating in-place, recovery issues can be solved much easier.

Temporal Database Management Systems. Keeping old versions comes at little extra cost in a log-only DBMS. Given a log-only DBMS, realizing a temporal DBMS should not add much extra cost.

Cache Coherence. Versioning/timestamping can be exploited in cache coherence protocols in client-server environments, as is done in BOSS [119].

Nomadic Computing. Objects and segments are timestamped. This can also be utilized to maintain consistency in client databases that are off-line part of the time. If these at regular times are connected to a server, they can be made consistent by uploading changes since the last connect.

The advantages listed above indicate that the log-only approach is highly interesting, but it should not come as a surprise that these advantages do not come for free. The log-only approach has similarities to other no-overwrite strategies, for example the shadow page approach, and it also inherits the nasty side of shadowing: after a while, data becomes unclustered. However, for several reasons, we expect that this will be less of a problem with our approach:

- The increased amount of main memory can to a large degree compensate for the lack of clustering.
- The access pattern is supposed to be more direct and navigational in an ODBMS than in a RDBMS.
- Some systems are mainly write-once systems (for example many SSDBs), and if a large batch is loaded at a time, we can get very efficient clustering.

It is also possible to *recluster* the database when needed, although this can be costly with large amounts of data. This can be done as a part of the cleaning process, which is performed asynchronously. While this at first glance might look as if we have to do twice the work to get the same result, compared to other systems, it is not necessarily so. If you use write-ahead logging (WAL), you also have to write the data twice, to the log as well as to the database itself. It is also important to remember that reclustering is also necessary in traditional DBMSs. Traditional clustering works well as long as the access pattern is static, but if the access pattern changes, the database have to be reclustered.

The cleaning process can also be utilized to do dynamic and adaptive clustering. With the advent of persistent programming languages that need efficient support for garbage collection, for example persistent Java, it is possible that the garbage collection can be done as a part of the cleaning process. In this way, the effective cost of the cleaning can be reduced.

In the case of a temporal DBMS, a kind of continuous reclustering is also needed in traditional systems. If you want to keep previous versions of data, and still want to keep the current set clustered, you have to move the old version before inserting the new.

5.3 Alternative Realizations

There are two alternative ways to realize a log-only ODBMS: *page-based*, and *object-based*. The most important difference between these two is how objects are indexed. In an object-based design, indexing is done at object granularity, with logical object identifiers, while in a page-based design, only the pages are indexed, and the page an object resides on, is hardcoded into its object identifier.

5.3.1 Page-Based Designs

In page-based designs, the log is seen as one large persistent address space. When an object is created, it is allocated space from this address space. The pages are written to the log, similar to the example in Figure 5.2. The objects are referenced by a persistent-memory address (a page identifier is included in the OID), and are retrieved via the page index which is interleaved in the log. If an object is modified, a new version of the page(s) it resides on is written back to the log.

The main advantage of the page-based approach is ease of implementation. However, it has some of the same problems as traditional page servers:

- Even if only a small part of the page is modified, the whole page has to be written back. If objects are not well clustered, this will give low effective write bandwidth.
- With bad clustering, main-memory buffer utilization will be bad as well.
- Reclustering is difficult, the indexing in a page-based design is similar to the use of physical object identifiers in a traditional system, even though the location of the pages in a log-only system changes, an object is bound to one page during its lifetime.
- Variable sized objects are difficult to integrate into the page approach, since the space is allocated when the objects are created. This makes it difficult to employ compression.

In addition to these well known problems from traditional page servers, a page-based log-only ODBMS also makes transaction management difficult. To avoid page level locking, you essentially need to have 1) an additional log to keep track of page updates, or 2) use ad-hoc techniques to solve the problem. Both solutions are likely to hurt performance and increase complexity.

Two page-based ODBMSs are Texas [189] and Grasshopper [91]. Based on the documentation of the commercial ODBMS MATISSE [134], it is also possible that MATISSE is page-based, but this has not been possible to verify.

5.3.2 Object-Based Designs

The alternative to a page-based design, is to index objects directly. In this case, only the object (or a delta object) needs to be written to the log when an object is modified. This is especially useful if good clustering is difficult. Dynamic clustering can be employed to give a good clustering. This is possible because clustering can change with time, according to the access pattern. It is also possible to get all of the other benefits of log-only systems, as described previously in this chapter.

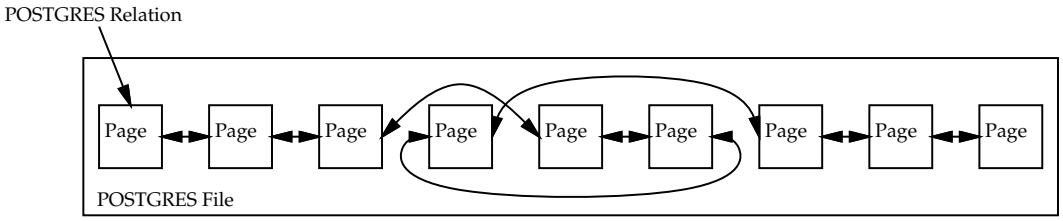


Figure 5.4: POSTGRES file.

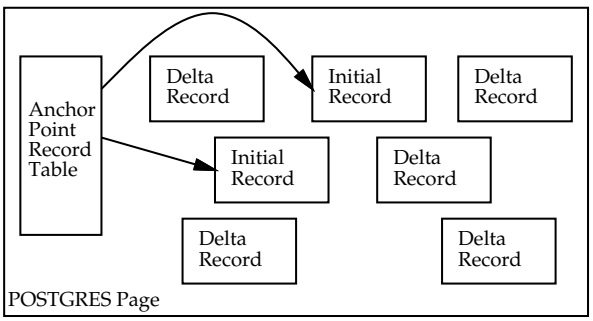


Figure 5.5: POSTGRES page.

The most important disadvantage with the object-based approach, is that more index updates might be needed, because individual objects are indexed, and not whole pages. We will later in this thesis develop techniques to minimize this disadvantage.

Based on the advantages and possibilities of an object-based design, we see it as the most interesting approach, and Vagabond, which will be described in detail in the rest of this thesis, is object-based. In the rest of this thesis, we write *log-only* as short for a log-only object-based design.

5.4 Systems Based on Log-Only Related Techniques

We will now present systems that employs log-only related techniques: POSTGRES, log-structured file systems (LFS), DBMS based on LFS, and the log-structured history data access method (LHAM). We also present other work, based on the ideas presented in these sections.

5.4.1 POSTGRES

No-overwrite strategies have a long history, for example in shadow-paging recovery schemes, like the one used in System R [5]. The best known log-only DBMS, and probably the first as well, was POSTGRES [195]. POSTGRES was an *extended relational database system*, which actually can be said to employ a no-log strategy rather than log-only.

Data in POSTGRES were stored in relations, which were stored in files. Pages were allocated or deallocated for a file on demand, and were linked together, as illustrated in Figure 5.4.

As illustrated in Figure 5.5, each page in POSTGRES had an *anchor table*, used to retrieve records stored on a page. When a record was created, space was allocated for the record. When records were

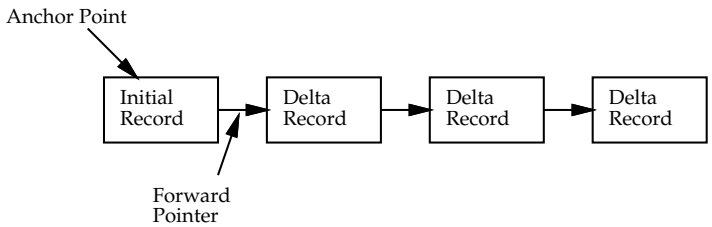


Figure 5.6: POSTGRES record.

updated, they were not updated in-place on the page, rather, a delta record was created, which recorded the changes from the previous version (illustrated in Figure 5.6). When a record was to be read, the whole chain from the first record had to be traversed and processed. POSTGRES was optimized for small records,[2] and delta records should be on the same page as the initial record.

Although POSTGRES introduced many novel ideas, the storage strategies did not gain much success at that time. The main reasons for this, were some serious problems resulting from the way records were stored:

- Read operations could be very expensive, because of the delta chains.
- POSTGRES used a *force buffer* policy. At commit, all data modified by the transaction had to be written, giving a very high commit cost.
- Even though POSTGRES could be used as a basis for a temporal DBMS, the use of append-only linked lists for each record was too inflexible and inefficient, an additional index was needed in most cases, increasing the overhead.
- In common with other no-overwrite strategies, POSTGRES also held the risk of declustering relations.

5.4.2 Log-Structured File Systems

The no-overwrite idea was borrowed from POSTGRES and used in log-structured file systems (LFS), first presented by Rosenblum and Ousterhout [177] in the Sprite LFS, and later refined by Seltzer et al. in BSD-LFS [185]. LFS has also been the basis for other systems, for example Spiralog [102, 212].

In an LFS, file and directory information is interleaved in a log. File identifying information is kept in inodes, similarly to Unix, and an inode map is used to locate the position of an inode in the log. It is assumed that the active portions of the inode maps can be kept in main memory. The granularity of writing (and indexing) in LFS is pages.

LFS has also been shown to be able to benefit from the advantages listed earlier in this chapter. It has been shown to be very well suited for tertiary storage management [69, 117], in on-line backup systems [78], and on-line data compression [36]. Implementing transactional support in LFS is described in several papers by Seltzer et.al. [182, 184].

The most important bottleneck in an LFS is cleaning, especially under heavy load, or when there is little free space on disk. This has been studied in several LFS performance improvement studies [19, 144]. This work has also resulted in improving the cleaning techniques and cleaning heuristics, and on self-tuning.

[2] A large object interface was added later.

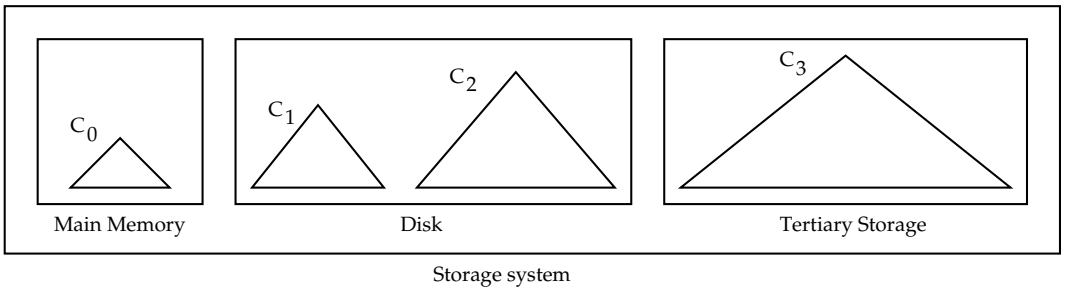


Figure 5.7: LSM with four components.

5.4.3 Log-Structured History Data Access Method

The *log-structured history data access method* (LHAM) [143, 170] is based on the *log-structured merge-tree* (LSM-Tree) [169].

A LSM is based on a hierarchy of index components, where the index component at each level has a larger size than the index component at the previous level in the hierarchy. We denote an index component C_i. Component C_i indexes a subset of component C_{i+1}, i.e., component C_i is (much) smaller than C_{i+1}. Component C_0 is in main memory,[3] $C_1 \ldots C_i$ are typically on disk and tertiary storage (see Figure 5.7).

Updates are only done to component C_0. Entries from C_0 not yet migrated to component C_1 are merged into component C_1 in batch, as a background process. In general, the same is the case for all the components in the hierarchy, entries from C_i not yet migrated to component C_{i+1} are merged into component C_{i+1} as a background process.

The most frequently updated entries will typically be in the lower levels of the hierarchy, because each update of an entry will result in an insertion into C_0.

As a result of the way updates are merged into the higher numbered trees, entries in component C_i are always as least as recent as entries in component C_{i+1}. When we search for an entry, we start the search in component C_0. If we do not find the searched entry there, we continue with component C_1, component C_2 and so on, until we find the entry, or have reached the last component. As long as we do the search as described, we are also sure to get the most recent version, even though the higher numbered components might contain outdated values.

Inserts and updates are only done to the first level index, and the contents of one level in the index are asynchronously migrated to the next level. As a result, all data inserted or modified during a certain time period will be in the same level. Search for data written at a certain time is efficient, but searching for the most recent version of certain data can be costly.

The main advantage of LHAM is support for high insertion rates, while also being competitive in terms of performance.

5.4.4 Other Related Work

Most no-undo/no-redo recovery approaches share some of the characteristics of the log-only approach. We have already mentioned the shadow-paging algorithm, used in System R. Other techniques are *differential files* (also called deferred update and side files), and the *database cache*. In the differential file approach, updates are done to a new and previously unused location, in a side file [77]. At regular

[3] Logging has to be used to be able to recover from main memory failure.

intervals, the contents from the side file are copied back to the original file. The database cache [63] uses a similar approach, but by assuming large main memory buffers, new algorithms can be used to avoid some of the problems of the shadow page and side file approaches.

Other relevant work includes approaches to deferred update, for example the BOSS approach [119]. BOSS employs WAL, but updates to the database itself can be deferred. The difference between deferred updates as used in BOSS and a log-only approach is that the log-only approach takes it to the extreme, there is *only* the log. The advantage with the log-only approach is that the updates to the database are avoided.

5.5 Summary

This chapter outlined the basic principles behind a log-only ODBMS based on LFS techniques. As described in the overview of other systems based on log-only techniques, the log-only approach is not new in itself, and even log-only systems can be said to be based on earlier techniques, for example shadow-paging.

The summary of advantages of a log-only systems should serve as a motivation to explore this strategy further, but it should be emphasized that whether a system might be able to achieve high performance has to be verified by analytical modeling or simulations. In Chapter 14 we will use analytical modeling to study the possible gain from using a log-only approach. However, real evidence can only be gained from an implementation of the approach, used in real applications. This is left as further work.

Part II

The Design of Vagabond

Chapter 6

An Overview of Vagabond

In this section, we briefly describe the architecture of the Vagabond temporal ODBMS, which is used as the context for the following chapters. The server architecture is described, and we give a summary of techniques that can be used to reduce the read costs in such a system. These techniques will be described in more detail in the rest of this thesis. We describe storage objects, and we describe how Vagabond can be incorporated into a parallel and distributed architecture. We emphasize that this is not the description of an implemented system, only a framework for the design presented in the rest of this thesis.

6.1 Server Architecture

Similar to the Shore ODBMS [39], we also use a peer-to-peer architecture. All application programs in the system are connected to one server, running on the same machine as the application. This server is the gateway to the DBMS, including remote servers (cf. the multiple client/multiple server architecture in Section 3.4). Not all servers have data stored locally. If all servers did, including those running on office workstations, that would make it impossible to achieve high availability. However, even if no data is stored locally, a server must be running on that node to make it possible for the application program to access the DBMS. One advantage of this approach is that it makes it possible for several clients running on the same machine to utilize a common server-side cache. On the client, client-side caching will be employed as well.

6.1.1 Client/Server Communication

A Vagabond server is an *object server* (see Section 3.6.1). The architecture of the server is illustrated in Figure 6.1. A client normally operates against the Vagabond API, a client-side stub which provides the mechanisms to communicate with the server. The communication with the server is done via the *messenger*.

6.1.2 Server-Side Operation

The server is multithreaded, and all subservers run as separate threads. There is also one thread for each transaction. To reduce thread creation costs, we use recyclable threads. Recycling of threads is done by having a fixed (but expandable) number of threads of a particular type. These threads are created when the server is started. When idle, they are waiting in an *thread pool*. When a task is to be started, it can be allocated one of the idle threads. When the thread has finished its task, the thread

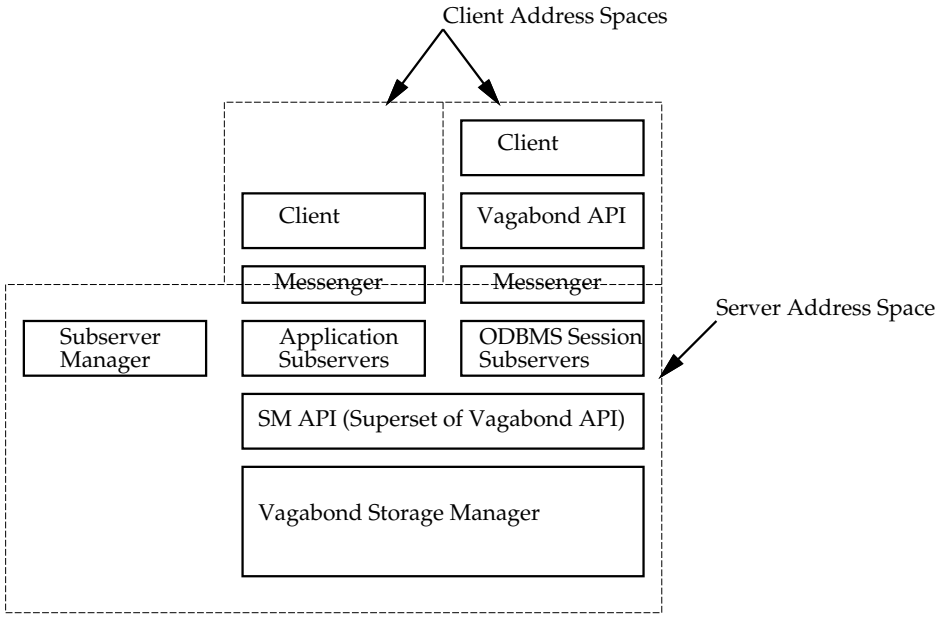


Figure 6.1: The Vagabond server.

is returned to the thread pool. This saves the overhead of creating threads, and the cleanup cost after thread termination. Even though the use of threads incurs extra cost compared with an event driven system, for example more locking overhead, thread-administration, and thread-switching overhead, multithreading makes it easier to exploit multiprocessor computers.

Each client that connects to the server starts the session by connecting to the *Subserver manager*, which allocates either an *ODBMS session subserver* thread or an *application subserver* thread to the client. The allocated thread operates in the server address space on behalf of the client. Commands and data are communicated through the *messenger*.

6.1.3 Subservers and Server Extensibility

All communication between clients and the storage manager is done via subservers. We have three classes of subservers:

- *Subserver manager*.
- *ODBMS session subservers*.
- *Application subserver*.

The subserver manager is only used when a session is started, to allocate the appropriate subserver, as described above.

ODBMS session subservers and application subservers access the storage manager (SM) through the SM API. Normally, ODBMS session subservers are used. Application subservers are extensions to the system, making it easy to extend Vagabond. This is similar to features found in other systems, for example the Value Added Server concept in Shore, and DataBlades/Cartridges in commercial systems. However, for such a concept to be really beneficial, the ODBMS has to be object shipping

rather than page shipping, so that the system is able to filter out objects and do operations on the objects, something which is impossible or difficult on page server systems.

One interesting point here, is that the SM API is a superset of the Vagabond API, the client interface stub. This feature makes it easier to implement and test subservers as clients, before they are added to the server. As subservers, they can communicate with clients through a messenger, as illustrated in Figure 6.1.

6.1.4 Storage Manager

The storage manager is responsible for permanent storage of objects. Its most important operations include transaction management, secondary and tertiary storage management, and indexing.

Buffering data in main memory is done to reduce the amount of data needed to be transfered between main memory and disk, and between individual servers. Important buffers include the object buffer, index node buffer, and object descriptor buffer (an object descriptor is an entry in the OIDX, see Section 3.1.2, and will be described in more detail in Section 8.1.2). These buffers can be dynamically resized, to get optimal performance with changing access patterns. The cost functions derived later in this thesis, and the papers in the appendixes, can aid in dynamically deciding optimal buffer sizes.

6.1.5 Permanent Storage

All data in a Vagabond server is stored in a logical log, which is stored in one logical *data volume*. A data volume consists of one or more storage devices. A storage device can be a secondary as well as a tertiary storage device. Typical examples of storage devices are:

- A raw disk partition (on magnetic disk).
- A fixed size (but extendible) file on the native file system simulating a disk partition. Running our own system on top of the native file system gives an extra level of indirection. However, allocating disk space on an (almost) empty disk will on most modern file systems give a mostly sequentially allocated file. This is done by creating a disk file, and writing as many blocks to it as the size specified. The file can be extended or shrunk by any integral number of segments.
- Optical disk.
- Tape.
- Flash memory.

Devices can be dynamically added to or removed from a data volume. Adding a device basically increases the number of available segments in the volume, while removing a device is done by first moving all data currently residing on that device to another device.

Even if disk space is cheap, it is still necessary for some applications to have data on tertiary storage. This can be done transparently in Vagabond. Tertiary storage is most often removable media, for example optical disk or tape, which can be used in disk and tape robots.

One of the devices in the data volume is called the *root device*. On this device, the most important volume information is stored, and it also contains the checkpoint block. The checkpoint blocks should be on a rewritable medium, so the root device will typically be a magnetic disk.

6.2 Objects in Vagabond

In Vagabond, all objects smaller than a certain threshold are written as one contiguous object. They are not segmented into pages as is done in other systems. Objects larger than this threshold are segmented into *subobjects*, and a *large-object index* is maintained for each of these large objects (this is done transparently for the user/application). There are several reasons for doing it this way:

- Writing one very large object should not block all other transactions during that time.
- A segmented object is useful later, when only parts of the object is to be read or modified.
- Parts of the object can reside on different physical devices, even on different levels in the storage hierarchy.

The value of the threshold can be set independently for different object classes. This is very useful, because different object classes can have different object retrieval characteristics. Typical examples are a video and an index. When playing a video, you want to retrieve one large segment of the video each time. On the other hand, when searching an index tree, you only want to retrieve single nodes, which usually have a small size. Similar for both video and index retrieval is that you only want a part of the object. For other objects, the whole object will be needed at once. One example is images. In order to be able to display the image, the whole object is needed. In that case, storing the image as one contiguous object will be advantageous.

Every object version in Vagabond has an associated object descriptor (OD), which contains the OID, physical location, timestamp, and other administrative information. In addition, every subobject has an associated subobject descriptor (SOD). ODs and SODs will be described in more detail in Section 8.1.2 and Section 8.5.2.

6.2.1 Typed Objects

It is not strictly necessary for the storage system to know the type of an object. Actually, in most systems, an object is simply a chunk of bytes from the storage manager's point of view, and page servers do not even have to know about objects, pages are all they care about. However, storing type information in the system can improve efficiency and performance considerably:

- It is useful for type checking.
- It makes it easier to employ hierarchical concurrency control techniques.
- If the server knows the attributes and attribute sizes of an object, it is easier to support vertical fragmentation in a parallel or distributed system. The alternative is that the application gives "partitioning hints", for example where in an untyped object it is possible to partition it. Some minimal information is also needed for reclustering and garbage collection (at least one needs to know where the pointers are).
- Typed objects are also necessary if the server shall be able to do some kind of server-side filtering or method execution.

In Vagabond, meta information for an object class or type is stored in the database as an object, called a *class descriptor object* (CDO). CDOs can be versioned as other objects, which simplifies support for class version management.

CID
Special object type
(Data field reserved for special object manager)
Maintain signatures?
Signature size
Large-object threshold
Subobject size
`NavigDesc` size
Vacuuming age
Metadata – Attribute information – Signature calculation information

Figure 6.2: Class descriptor (CDO).

Every object in the system belongs to a "class" (not only a programming language class, for example, it can also be an index class) as described in a CDO. A CDO is uniquely identified by a *class identifier* (CID), and the CID is used in object descriptors (ODs) to identify the class an object belongs to. The structure and contents of the CID are discussed further in Section 8.1.2.

The structure of a class descriptors object (CDO) is summarized in Figure 6.2. In the case of objects to be handled by special object handlers (see Section 6.2.4), the *special object type* identifies which special object handler should be used. The *reserved field* is reserved for the special object handler, and can be used to identify index variants, for example different key types in the case of an index. The `NavigDesc` *size* is the size of the *navigational descriptor*. A navigational descriptor exists in some index objects, for example B-trees, where it is a `(key,pointer)` tuple. The use of the `NavigDesc` will be further explained in Section 8.5.2 and Chapter 10.

The *large-object threshold* can be set to different values for different object classes, making subobject partitioning very flexible. The *subobject size* is the size of the subobjects in a large object (this can be different for different classes).

The *vacuuming age* is used for lazy vacuuming (see Section 12.6). If the timestamp of the object is older than the vacuuming age, the object can be removed. The default value of this attribute is a null value, i.e., the objects in this class can not be vacuumed.

The CDOs can hold other associated meta information as well, typically attribute and value offset information. This is also the place to store which attributes should be used to create the object signature if that is enabled for the particular class (hash-based signatures can be used to reduce the number of objects that need to be retrieved from disk, this will be described in more detail in Section 7.1). Whether to maintain signatures or not, is defined by the *maintain signatures* field. The size of the signature is stored in *signature size*. Note that using a *signature size* of zero to imply that no signature should be maintained, in stead of using a *maintain signatures* field, is not possible. The reason is that we want to make it possible to later disable signature maintenance for a class, without changing the size of the ODs, which include the signature. If this functionality was implemented by setting the signature size to zero, we would have to reorganize the relevant part of the OIDX to reflect the changed size of the ODs.

The CDOs are stored in the log as objects. When a new class is created, the class descriptor object is stored on all participating servers (full replication). The number of classes is in general small, so

the space used for this information will not represent a problem. Additionally, the information in a CDO will be frequently used, and it is therefore beneficial to have this resident at all servers. Creation of classes is an infrequent operation, compared to object creations, and the replication of CDOs will not represent a performance problem. It is not likely that an application needs real time response to class creations.

6.2.2 Temporal Aspects

Our storage structure is intended to be suited as a basis for a temporal DBMS. We maintain the temporal information in the index, which makes retrieval efficient even without additional temporal indexes.

6.2.3 Isochronous Retrieval

Some applications, for example video servers, do not want all of the objects delivered at once. Rather, they want part of it delivered at an appropriate rate, *isochronous retrieval*. One possible strategy to solve this is two queues in the I/O system. One for "normal" data, and one for high-priority audio/video data.

6.2.4 Special Objects

A large object can be viewed as an array of bytes, and retrieval of part of the object is done by retrieving a certain byte range of the object. This is not flexible enough for some of the structures that are stored as large objects, for example indexes. These structures are stored as large objects, but the subobject index has additional information to support more complex indexes. They can also have different concurrency control and recovery characteristics. These objects, which we call *special objects*, are handled by *special object handlers*, which will be treated in more detail in Chapter 10.

6.2.5 Examples of Special Objects

Class descriptor objects, persistent roots, collections, index structures and spatial data structures are also stored as ordinary data objects. This has the advantage of making them an integral part of the object system.

Collections

A collection is a collection of objects, for examples a set, bag, array or list,[1] with supporting methods for inserting, removing, and testing for the existence of a certain element. It also supports the use of an iterator to access the elements of a collection.

Secondary Indexes

To make an ODBMS efficient, we need secondary indexes in addition to the OIDX. One-dimensional as well as multi-dimensional indexes, which can be suitable for temporal queries as well as for spatial data, should be supported.

In Vagabond, indexes are also supposed to be stored as objects. An index will often be a large object. In many systems, all indexes have the same index node (block) size. In Vagabond, this can

[1] Collections are in some literature also called *containers*.

be tailored, so that different indexes, for different object classes, can have different index node sizes, depending on expected and actual access patterns.

To be able to use this system as a basis for a GIS, it is necessary to have support for spatial data structures. Only very recently have commercial DBMS with spatial support emerged, some with the data structure implemented in BLOBs, other with more integrated extensions, such as Informix Universal Server's geodetic DataBlade module, DB2's Spatial Extender, and Oracle 8i/Spatial Cartridge.

Most ODBMS vendors do not have the infrastructure or the architecture necessary to support scalable spatial data management, and a client-side index solution is necessary. One example of this is ObjectStore.

Persistent Roots

To be able to access the objects later, we need some handles into the database. This is typically done by the use of *persistent roots*. A persistent root is a *named object*, i.e., a tuple consisting of a name (a string), and an OID. The persistent roots are stored in a persistent root object, which is an object with a predefined OID. The persistent root object itself is an index.

Multidimensional Arrays

The storage scheme we have described here is particularly applicable to arrays, which are heavily used in scientific computing. Subarrays are stored as contiguous chunks in the segments, which will give very good performance, even for read-only transactions.

6.3 Read and Write Efficiency Issues

The system is write-optimized, and as a result, object retrieval and index lookup can become a serious bottleneck. We employ some techniques to reduce the number and size of read operations. These techniques can improve performance considerably, with none or marginal write penalties:

- Careful layout of objects.
- Hash-based signatures.
- Clustered index.
- Object compression.

The last one, object compression, will also improve write efficiency, as it reduces the amount of data that needs to be written to disk. In addition to the techniques listed above, we also employ writing of delta objects to further reduce the amount of data needed to be written to disk (this technique reduces the write cost, but increases read cost), and a number of index optimizations as will be discussed in Chapter 8 and Chapter 9.

6.3.1 Careful Layout of Objects

Several strategies are used to store objects on disk in a way that reduces read cost. One important strategy is to try to store related objects close to each other when objects are stored on disk. Because we employ a no-overwrite strategy, heuristics can be used to reorder objects in segments that are to be written to disk, and during cleaning of segments.

Another possible strategy is to use heuristics to arrange objects in a segment so that they do not span more disk blocks than necessary (boundary alignment and reordering). In this way, a minimal number of disk blocks needs to be read when data is retrieved from disk.

6.3.2 Signatures

We employ a technique similar to signature files to reduce the number of objects that needs to be retrieved. This can be done with a very small extra cost in Vagabond. This is described in detail in Section 7.1.

6.3.3 Clustered Index

The OIDX is organized in a way that clusters OIDX entries for object belonging to a physical container (a collection of related objects, to be described in more detail in Section 8.1.1). A physical container can for example be used to store all objects from a class (and implicitly maintaining class extents), or all objects in a collection. This can makes set-based queries more efficient. If using signatures as well, the actual number of retrieved objects can in many queries be very low, as we in this case only have to scan the relevant part of the index and the objects with matching signatures.

6.3.4 Object Compression

To further reduce storage space, and disk bandwidth, objects can be compressed before they are written. This is described in Section 7.2. With a log-only approach, objects are written to a new location every time, so that we only use as much space as the size of the current version that is written.

6.3.5 Delta Objects

Often, only a small part of an object is changed when a new version is created. In this case, much can be gained if only the changes are written. This is especially the case if an object is a write hot spot object. An object that only contains the changes from the last version of the object, is called a *delta object*. Unlike traditional systems, that only use delta objects to reduce the log writing, a delta object in a log-only database system can be an object version on its own, i.e., the complete version will not necessarily be written.

The delta object itself can be made at a low cost, for example by using the following algorithm:

1. Do a bitwise XOR on the new and the old version of the object. The resulting bit string will now have 1's only in positions where there is difference between the old and new version.

2. The resulting bit string can be run-length encoded, and the resulting delta object will be very small if there has been only small changes between the two versions.

This algorithm is most beneficial when the objects have the same object length and fixed size attributes. The algorithm can easily be extended and improved, for example by only considering updated attributes.

In general, generating and writing a delta object is only relevant if the previous version is already in memory. This is usually the case. If not, an inefficient installation read of the previous version would be needed to be able to generate the delta object.

It is not always beneficial to write delta objects in the case of large objects. Many large objects are relatively static objects, and when updates are done, large parts of the affected subobjects (a large

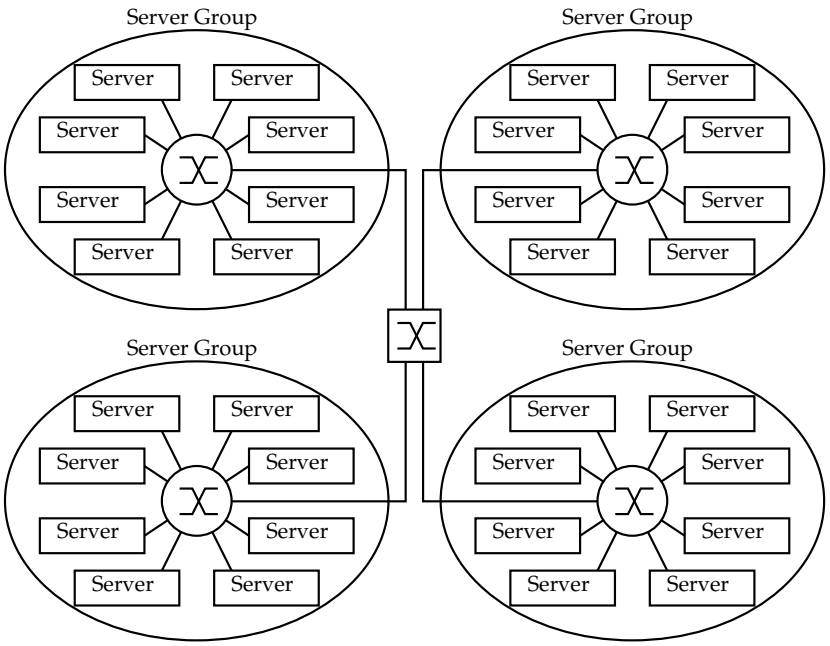


Figure 6.3: Vagabond system architecture.

object is physically partitioned into subobjects) are modified. Other large objects, for example index structures, are usually more dynamic, and updates only affect a small part of one subobject. For such objects, writing delta objects is beneficial.

The disadvantage of writing only a delta object is, of course, that previous versions have to be retrieved to reconstruct an object at read time. This problem can be reduced by writing the complete version of an object when an object is to be replaced in the buffer due to buffer replacement policy (this is done in addition to writing delta objects, i.e., if an object has been updated several times before it is discarded from the buffer, several delta objects might have been written). In this way, reading can be done efficiently later (note that because of the log-only strategy, this writing is relatively cheap). Reading the chain of objects is only needed if the DBMS has crashed before a non-delta object has been written. This strategy is most beneficial for non-temporal objects. In the case of temporal objects, we also need to reconstruct previous versions, and not only the most recent version. The described strategy does not solve that problem, but the problem can be reduced by writing complete versions at regular times, for example a full object version for every n'th delta version.

6.4 Parallelism and Distribution in Vagabond

The Vagabond architecture is a system designed for high performance, and one strategy to achieve this, is to base the design on the use of parallel servers. Objects are declustered over a set of servers, which we call a *server group*. The declustering is done according to some declustering strategy, this is further discussed in Section 11. The servers in a server group can cooperate on the same task, and in this way, it is possible to get a data bandwidth close to the aggregate bandwidth of the cooperating servers. To benefit from the use of a parallel server groups, it is supposed that the servers in one server group are connected by some kind of high speed communication.

The demand for support of *distributed databases* is increasing, and to satisfy this, we use a hybrid solution: *a distributed system, with server groups* (Figure 6.3). In this way, objects are clustered on server groups based on locality as is common in traditional distributed ODBMSs, but one server group can contain more than one computer (a kind of "super server").

6.5 Summary

This chapter described the overall architecture of the Vagabond ODBMS, and the main features. The rest of this thesis will concentrate on *how* to make a system that can deliver support for these features. We will identify potential bottlenecks, and describe how the impact of these bottlenecks can be reduced.

Chapter 7

Reducing the Data Transfer Volume

In this chapter we describe two techniques that can be used to reduce the amount of data transfer between memory and disk as well as between different servers: signatures and data compression. The techniques are well-known, but previously, only limited success has been achieved from using these techniques. However, some of the factors that previously have reduced their practical application are not present in a log-only system, making the gain from using these techniques larger in a log-only system than in a system based on in-place updating.

7.1 Signatures

A signature[1] is a bit string, which is generated by applying some hash function on some or all of the attributes of an object. By applying this hash function, we get a signature of F bits, with $m \leq F$ bits set to 1. If we denote the attributes of an object as $A_1, A_2, \ldots, A_n$, the signature of object i is:

$$s_i = S_h(A_j, \ldots, A_k)$$

where S_h is a hash value generating function, and $A_j, \ldots A_k$ are some or all of the attributes of the object (not necessarily including *all* of $A_j, \ldots, A_k$, S_h does not necessarily use all its arguments). Similar to hashing in general, two objects with the same signature may or may not have the same (shallow) value, *but objects with different signatures are guaranteed to differ*. The size of the signature is usually much smaller than the object itself, and it has traditionally been stored separately from the object, in a signature file.

When searching for objects that match a particular value, it is possible to decide from the signature of an object whether the object is a possible match. By first checking the signatures when doing a *perfect-match query*, the number of objects that has to be retrieved can be reduced. This can considerably reduce the total retrieval cost, because the size of the signature file is smaller than the total size of the objects involved in the query.

A typical example of the use of signatures is a query Q to find all objects in a set where the attributes match a certain number of values:

$$A_j = v_j, \ldots, A_k = v_k$$

This can be done by calculating the query signature s_q of the query:

[1] Note that the term *signature* is also used in other contexts, e.g., function signatures and implementation signatures.

$$s_q = S_h(A_j = v_j, \ldots, A_k = v_k)$$

The query signature s_q is then compared to all the signatures s_i in the signature file in order to find possible matching objects. A possible matching object, a *drop*, is an object whose signature s_i is equal to s_q (in the case of signatures generated by superimposition, which will be discussed below, a drop is a signature where all bit positions set to 1 in the query signature are set to 1 in the object's signature). The drops form a set of candidate objects. An object can have a matching signature even if it does not match the values searched for, so all candidate objects have to be retrieved and matched against the value set that is searched for. The candidate objects that do not match, i.e., objects with the same signature as the query signature, but not matching the query, are called *false drops*.

Signature files have previously been shown to be an alternative to indexing, especially in the context of text retrieval [15, 66]. They can also be used in general query processing, although this is still an immature research area. The main drawback of signature files, is that signature file maintenance can be relatively costly; every time the contents of an object change, the signature file has to be updated as well. To be beneficial, a high read to write ratio is necessary. In addition, high selectivity is needed at query time to make it beneficial to read the signature file in addition to the candidate objects.

We will now describe in more detail how signatures are generated, signature storage alternatives, and how signatures can be used in an ODBMS without requiring a high read to write ratio.

7.1.1 Signature Generation

The methods used for generating the signature depends on the intended use of the signature. We will now discuss some relevant methods.

Whole Object Signature

In this case, we generate a hash value from the whole object. This value can later be used in a perfect-match search that includes all attributes of the object. This method is only useful for a limited set of queries, where all the attributes of the object are involved in the perfect-match search.

One/Multi-Attribute Signatures

A more useful method is to computer the hash value of only one attribute of the object. This can be used for perfect-match search on a specific attribute. Often, a query is on perfect match of a subset of the attributes, similar to the example above. If such queries are expected to be frequent, it is possible to generate the signature from these attributes, again only looking at the subset of attributes as a sequence of bits. This method can be used as a filtering technique in more complex queries, where the results from this filtering can be applied to the rest of the query predicate.

The one/multi-attribute signature method is not very flexible, as it can only be used for queries on the exact set of attributes used to generate the signature. In the case of small sized attributes in a traditional system, an index would in general be more suitable. Its search performance will be better, and it supports range queries. In the case of large attributes, it is possible to use the signature instead of the whole attribute in the index. Using one/multi-attribute signatures when these signatures can be embedded into the OIDX, can still prove to be beneficial (see Appendix F).

Superimposed Coding Methods

The real power of signatures comes when the *superimposed coding* technique is used. When this techniques is employed, we get a signature that can be used for different perfect-match queries, where the different queries involve different sets of attributes.

When superimposed coding is used, we first compute a separate attribute signature $S_h(A_i)$ for each attribute in the object. The object signature itself is generated by performing a bitwise OR on each attribute signature. For example, for an object with 3 attributes, the object signature is calculated as:

$$s_i = S_h(A_0) \text{ OR } S_h(A_1) \text{ OR } S_h(A_2)$$

This results in a signature that can be very flexible in use, we can do a perfect-match search on any subset of attributes. When comparing a search signature with object signatures generated by superimposed coding, an object is a drop if all bit positions set to 1 in the query signature are set to 1 in the object's signature. It is also possible that other bit positions in the object's signature are set to 1, but that is not relevant for the actual query. The other bits set to 1 have been set as a result from attributes not part of the query.

Superimposed coding can also be used on set-valued attributes (a set-valued attribute is an attribute that itself is a set). In this case, a signature is generated for each member of the set. These signatures are OR-ed together to generated the attribute signature [96, 114]. By using this technique, queries of the type *is-subset*, *has-subset*, *has-intersection* and *is-equal*, can be answered efficiently, in many cases with less cost than alternative methods, for example using nested indexes.[2]

7.1.2 Signature Storage

Traditionally, the signatures have been stored in separate files, outside the indexes and objects themselves. A signature file contains the signatures s_i for all objects i in the relevant set. The size of a signature file is in general much smaller than the size of the relation/set of objects that the signatures were generated from, and a scan of the signature file is much less costly than a scan of the whole relation/set of objects. The most well-know storage structures for signatures are *Sequential Signature Files* (SSF) and *Bit-Sliced Signature Files* (BSSF), which are most suitable for relatively static data [66]. To better support inserts, deletes, and updates, several dynamic signature file methods have been proposed, based on multi-way trees and hash files.

Sequential Signature Files

In the simplest signature-file organization, SSF, the signatures are stored sequentially in a file. A separate *pointer file* is used to provide the mapping between the signatures and the objects. In an ODBMS, this pointer file will typically be a file with OIDs, one for each signature. During each search for perfect match, the whole signature file has to be read. When an object is updated, one entry in the signature file needs to be updated.

Bit-Sliced Signature Files

With BSSF, each bit of the signature is stored in a separate file, so that with a signature size F, the signatures are distributed over F files, instead of one file as in the SSF approach. This is especially

[2] A nested index is a B-tree variant where the leaf node entries are composed of a key value and the OIDs of the objects that have this key value in the indexed attribute [14].

useful if we have large signatures. In this case, we only have to search the files corresponding to the bit fields where the query signature has a "1". This can reduce the search time considerably. However, each update implies updating up to F files, which is expensive. So, even if retrieval cost has been shown to be much smaller for BSSF, the update cost is much higher. Thus, BSSF based approaches are most appropriate for relatively static data.

Several improvements of the BSSF have been proposed, most of them imply some vertical or horizontal decomposition [87, 113, 172]. Variants that use signature compression and multi-level signatures also exist.

7.1.3 Signatures for Fast Text Access.

Fast text access has been the main application of signatures, and most of the publications on signatures have been related to text access methods [15, 65, 66, 120, 206, 217]. In this case, the signature is used to avoid full text scanning of each document, for example in a search for certain words occurring in a particular document.

Documents are first divided into logical blocks, which are pieces of text that contain a constant number of distinct words (if most documents are small and have approximately the same size, this step is not strictly necessary). A separate signature is generated for each of these logical blocks, i.e, there is in general more than one signature for each document. In order to generate a block signature, a word signature is generated for each word in the block, and the block signature is generated by OR'ing these word signatures.

When searching for documents containing one or more particular words, the signature file is read first, and each block signature is compared with the query signature (the signature generated from the query words). This gives us a set of candidate documents (or candidate blocks), where the actual search words might occur. These documents have to be retrieved and searched.

Example

To illustrate the advantage of using signatures for fast text access, consider a collection of 1024 technical documents. The average document has a size of 64 KB, and contains 600 distinct words. Without signatures (or an index), all documents have to be retrieved if we want to find which documents contain one or more specific words. The total data volume to read will be $1024 * 64$ KB$= 64$ MB.

The data volume to be read can be reduced if signatures are employed. In this example, assume a signature size of $F = 4096$ bits, and that these are stored in an SSF. We do not divide the documents into logical blocks. The size of the signature file will be $S = 1024\frac{F}{8} = 1024 * 512 = 512$ KB.[3]

When searching for documents containing one or several words, we first read the signature file. For all documents with matching signature, we have to read the document. The probability that a retrieved document does not contain the actual word, is equal to the false drop probability [66]:

$$F_d = (\frac{1}{2})^m, \text{where } m = \frac{F \ln 2}{D}$$

In the case of a text document, D is the number of distinct words in a block. In this example, only $F_d = 0.037$, i.e., 3.7% of the retrieved documents, will be false drops. Thus, instead of reading 64 MB, we can satisfy the query by only reading the signature file and the candidate documents. The number of documents to retrieve depends on the selectivity of the query. If we search for a very

[3] If a document identifier is included in the signature file, its size will be slightly larger than this.

common word, most of the documents have to be retrieved, but if we search for a combination of words, the number of documents to retrieve will in most cases be low.

How to find the optimal signature size is an issue when using signatures, and the size depends on several factors, including the number of candidate documents. A large signature reduces the false drop rate, but increases the size of the signature file, which has to be read in its entirety for all queries.

In the example above, we assumed that the signatures were stored in an SSF. Another alternative is to use BSSF. In this case, the signature files would occupy the same amount of disk space, but on average, only half of them had to be read to answer a query. This might at first seem like an advantage, but in practice, accessing extra files implies large overheads, so BSSF would not be beneficial with a small number of signatures as in this example.

7.1.4 Storing Signatures in the OIDX

In a write-optimized system, object retrieval can become a bottleneck. This bottleneck can be reduced by including the object's signature in the OIDX. In Vagabond, the OIDX is updated every time an object is modified, and if we store the signature in the object's object descriptor (OD) in the OIDX, the additional signature maintenance cost is only marginal. This is different from traditional systems, where the signature file has to be updated every time an object is updated, reducing its effect. In those systems, a large read to update ratio is necessary if the use of signature should be beneficial.

Perfect-match queries can use the signatures in the OIDX to reduce the number of objects that have to be retrieved, as only the candidate objects, with matching signature, have to be retrieved. When the signature is stored in the OD, scan queries can be done efficiently by simply doing a scan over the relevant part of the OIDX, and only the candidate objects need to be retrieved. Because the OD is accessed on every object access in any case, the additional signature-retrieval cost is only marginal.

Optimal signature size is very dependent of data and query types. In some cases, we can manage with a very small signature, in other cases, for example in the case of text documents, we want a much larger signature size. It is therefore desirable to be able to use different signature sizes for different object classes. In any case, we have a tradeoff between signature size and additional signature-maintenance cost. Even though a small signature has only marginal effect on OIDX access cost, using larger signatures will increase the cost to a significant level.

The maintenance of object signatures implies computational overhead, and is not always required or desired. Whether to maintain signatures or not can be decided on a per class basis. This is also the case with which attributes to use when calculating the signature. This information is stored separately for each class, in the class descriptor object (see Section 6.2.1). To avoid complex index node management, all ODs in a physical container have the same signature length.

A more detailed study of performance aspects of storing signatures in the OIDX is presented in the paper included in Appendix F.

7.1.5 Signature Caching

The signature could also be stored together with the object on disk. In this way, the additional update cost is small. This is obviously of no use if the signature is discarded from the buffer at the same time as the object is discarded. However, it is possible to store the signatures of frequently accessed objects in a *signature cache*. Because the signature size is small compared to the object size, reducing the number of objects that fit in the object buffer and instead using the memory for buffering signatures, can improve the performance.

The signature cache approach is particularly interesting for page server ODBMSs using physical OIDs, and in [156] we have shown that in such systems the average object access cost can be significantly reduced by the use of a signature cache. Signature caching can also be used in order to reduce the communication costs in a parallel ODBMS [154].

7.2 Object Compression

To reduce storage space, as well as the amount of disk bandwidth used, objects can be compressed before they are written to disk. Compression/decompression can be transparent to applications, which means that compressed objects are decompressed by the server before they are made available to the applications. In many application areas, for example SSDBs, it is typical to have objects (or tuples) with a very large number of attributes, of which many of them have null values. By compressing these objects, it is possible to reduce both storage space and read/write disk bandwidth. Another example application is text, which can usually be compressed down to less than 50% of the uncompressed size.

The idea of compression in databases is not new, and some work exists, especially in the context of SSDBs. A more general study and overview of support for compression of data in databases has been given by Iyer and Wilhite [98]. They also analyze different design options with different data sets.

In traditional systems, compression has been difficult to employ efficiently. The reason for this is that the effective compression ratio changes with the contents of the object, so that different versions of an object can have very different sizes after compression. As it is impossible to know the size of future versions of a compressed object, it is necessary to reserve as much space as the maximum size of a compressed object when updating in-place. When using a log-only approach, an object is written to a new location every time. In this way, a version only needs to occupy as much space as the size of the compressed version.

Better compression can be achieved if knowledge of the structure of the objects is available. One example of how easy this can done, is the use of a bit mask for each object, with one bit for every attribute of the object. A bit is set to zero if the attribute is null, if not, it is set to one. In this way, attributes with a null value need not to be stored at all. A variant of this technique, a descriptor containing the offset of the start of each non-null field, was used in System R [5] and POSTGRES [199].

Even without knowledge of the object structure, good results can be achieved. In this case, objects are simply treated as bit streams. To avoid using too much CPU resources, a low cost compression algorithm should be used, for example run-length encoding.

The fact that compressed data have to be decompressed before used, implies that queries accessing large amounts of data only to check for a match in one or more attributes can be costly. Without compression, such queries are usually I/O bound, but can easy become CPU bound if data is compressed. By combining signatures and compression, this problem can be reduced. When signatures are maintained, perfect-match queries on attributes in large sets of objects can be done efficiently even without decompressing the objects.

7.3 Summary

In this chapter we have described the use of signatures and data compression. In the past, these techniques have only had limited success in DBMSs. However, we expect that they can be more beneficial in a log-only system than in an system based on in-place updating, and in particular, when used together in such a system.

Chapter 8

Object-Identifier Indexing

In an ODBMS, an object is uniquely identified by an object identifier (OID), which is also used as a "key" when retrieving the object. As discussed in Section 3.1, OIDs can be physical or logical. In a log-only ODBMS, objects are never written back to the same place. This means that logical OIDs have to be used, and an OID index (OIDX) is necessary. The number of objects in a database can be very large, and a fast and efficient index structure is necessary to avoid OID indexing becoming a serious bottleneck. This chapter describes an object-index structure suitable for indexing OIDs in a temporal ODBMS, and provides algorithms for efficient access to the index.

8.1 Contents and Structure of the OID Index

The OIDX contains the necessary information to map from a logical OID to the physical location where the object is stored. The physical location, together with the timestamp and other administrative information, are stored in index entries which we call *object descriptors* (ODs). A new OD will be created for each new version of an object, so that for each object, there can be more than one OD in the OIDX, corresponding to the number of versions of the object.

In general, an ODBMS can manage multiple logical databases. The logical databases can be represented as one or several *physical* databases. In Vagabond, we use one physical database for each logical database, and each logical database has a separate index.

An OID is only unique inside one database, thus, object identifiers in different databases will represent different objects. All database sessions are performed against a certain database, which database to access is given implicitly, and it is not necessary to contain database identifying information in the OID.

As discussed in Section 3.1.2, the OIDX in a traditional system is usually realized as a hash file or as a multi-way tree structure. In a log-only system, a tree structure is the only reasonable alternative, since an index node will be rewritten to a new location every time it is updated. If a hash file was used, an additional tree index on top of it would be necessary. We would then effectively end up with a tree structure anyway. The same is the case if we wanted to use the direct mapping technique. For this reason, all index structures considered in this chapter are variants of multi-way trees.

We will in the rest of this section describe the structure and contents of the OIDs and ODs in Vagabond.

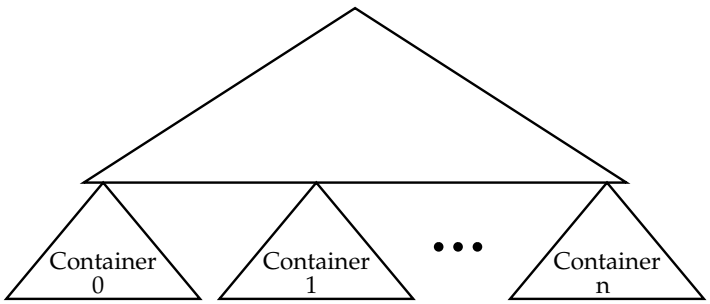


Figure 8.1: OIDX with containers.

8.1.1 Object Identifiers in Vagabond

In Vagabond, an OID is composed of three parts:

1. *SGID:* Server group identifier. The *SGID* is the identifier of the server group (see Section 6.4) where the object was created.

2. *CONTID:* Container identifier. The *CONTID* identifies the container the object belongs to (see below), similar to a file in other DBMSs.

3. *USN:* Unique serial number. Each object created on a particular server *SGID* and to be included in container *CONTID* gets a *USN* which is one larger than the previous *USN* allocated in this container.

The reasons for including a *SGID* and an *USN* in the OID should be obvious, but the rationale behind the *CONTID* needs some further elaboration.

In many page server ODBMSs, the objects are stored in containers, also called files, or relations (for example Objectivity, see Section 3.1). Which container to put an object in, is decided when the object is created, and this decision is often made according to some clustering strategy (see Section 3.3). In a traditional ODBMS, a part of the OID is often used to identify in which container the object is stored (see Section 3.1).

To benefit from the log-only approach, objects can not be stored in distinct files or clustered together in the same way as is beneficial in a system using in-place updates. This would make it difficult to achieve long, sequential writes. However, an approach similar to physical clustering of objects can be used for the OD in the OIDX. Similar to the way object clustering in page servers reduces the number of pages to read and update, clustering together ODs that are expected to be accessed together close in time, will reduce the cost of OIDX accesses. This is achieved by associating every object in a database with a container (see Figure 8.1). Which container an object belongs to, is encoded into its OID.

Note that the storage of objects is independent of which containers they belong to (for example, there is no relation between a segment and a container), the use of containers is only a way to cluster related ODs.

An object can be migrated from one container to another. If this is done, forwarding information is stored inside the OD representing the current version in the original container. When a migrated object is to be retrieved, two OIDX lookups are needed in order to retrieve the OD: one lookup in the original container, and one lookup in the container the object currently belongs to.

Field	Size (bits)
OID:	
SGID	32 (Only present when outside OIDX nodes)
CONTID	32 (Only present when outside OIDX nodes)
USN	64
Physical location	64
Object size	32
Create timestamp	64
End timestamp	64 (Only present when outside the OIDX)
Class identifier (CID)	24
Delta object?	1
Large object?	1
Temporal object?	1
Compressed object?	1
Inlined object?	1
First version?	1
Migrated to another server group?	1
Migrated to another container?	1
(Signature)	Optional field, variable size

Table 8.1: Contents and size of fields in the object descriptor.

In addition to using the containers as a way to cluster the ODs of objects that are expected to be accessed together, but in other ways are unrelated (i.e., different classes), they are also useful as a way to realize logical collections of objects from the same class, for example sets/relations, bags and class extents.[1] For example, one collection can be stored in one container. When this is done, scans and queries against these collections can then be executed efficiently. When using signatures as well, it will for many queries only be necessary to read a small proportion of the objects.

Another interesting use of containers is to have more flexibility in deciding the length of the search path for ODs. This can be achieved by storing hot-spot objects in small containers (i.e., containers with only a few objects) to get shorter search and update paths.

8.1.2 Object Descriptor Structure

The contents of an OD are summarized in Table 8.1, together with the size of the individual fields (Fields occupying one bit are used for boolean values).

The ODs are stored both in the OIDX and together with the objects in the segments. The reason for storing them in the segments as well, is to help identifying objects during cleaning, and it also works as a kind of write-ahead logging of ODs, in order to avoid synchronous updates of the OIDX at commit time.

The information in the OD gives a high degree of flexibility and efficiency, and even though it contains many fields, most of them can be stored in a compact way, many of them occupying only one bit. As will be described later in this chapter, the *SGID* and the *CONTID* are given implicitly

[1] A class extent is a collection of all the objects of a certain class in a database.

when stored in the OIDX, so they need not to be stored. In addition, the *USN*s of the ODs stored in one index node will be from a limited integer range, so that prefix compression can be used. When in main memory, and outside an index node, the *CONTID* must also be included in the OD. ODs for objects from other server groups are treated as a special case, so that the *SGID* is only used for the "remote" ODs. We will now describe in detail the contents and function of these fields.

Physical Location

This is the location of the object in the log. If the object is a large object, this location is actually the location of the root of the subobject index of the object (see Section 8.5.2). If the physical location is NULL, but the OD contains a valid timestamp, this OD is a tombstone OD (the object is deleted, but previous versions exist).

If an object is moved, for example during cleaning, the physical location in its OD has to be updated.

Object Size

All objects smaller than a certain threshold are written as one contiguous object, while objects larger than this threshold are segmented into subobjects, and a large object index is maintained for each of these large objects. The *object size* field in the OD is the size of a small object when in the log.

If the object is compressed, the actual size can be larger the *object size*. In the case of fixed-size objects, the size of the uncompressed object can be found from the class-descriptor object. If it is a variable-sized object, the size is stored together with the compression information in the compressed version of the object.

In the case of large objects, the object-size field is not used (if necessary, the object size can be found from the subobject index). Note that the fact that only 32 bits is used to store the object size only restricts the maximum size of a "small object", large objects can be larger than this.

Create Timestamp

This is the commit time of the transaction that created this version. Each transaction needs distinct timestamps, so a very fine timestamp granularity has to be used. A 64 bit timestamp is more than what is actually needed, but a 32 bit timestamp is not sufficient, and using a size between 32 and 64 bits is not efficient.

Timestamps with the most significant bit set are reserved for the case when a transaction identifier is used instead of the timestamp in the OD. This will be further explained in Chapter 12.

End Timestamp

The end timestamp is the time when the next version was created, or the time of deletion in case there are no new subsequent versions. The create and end timestamps give the interval an object was valid. If the OD is the OD of the current version of an object, the end timestamp is NOW, which is represented by the value NULL.

When in the OIDX, the end timestamp is given implicitly from the create timestamp in the OD of the next version of the object. This OD will in most cases reside on the same index node, so it is not necessary to store it here. However, when the OD is outside the OIDX,[2] the end timestamp is

[2]This also includes the PCache, to be presented in Chapter 9, where the ODs also include the end timestamp.

included. This makes certain operations and buffering of ODs more efficient (see Section 12.2.5).

Delta Object

Delta object is set to true if this is a delta object (see Section 6.3.5).

Large Object

This is true if this version of the object is a large object. An object can be a "plain data object" as well as a special objects (for example an index, as described in Section 6.2). If it is a special object, the relevant information is stored in the object's class descriptor (CDO).

Temporal Object

Temporal object is set to true if this is an object where we want to keep old versions when the object is modified or deleted. This is decided for each object at object creation time, but can be changed later (although this is not always a good idea, for example, this must be done with care with respect to cleaning).

It should be noted that this information could also be stored in the class-descriptor object. In that case, all objects in a class are either temporal or not. Which approach to use depends on whether orthogonality with respect to temporality is desired or not.

Compressed Object

In many cases, it is worth using some extra CPU cycles to try to reduce the size of an object before storing it in the log (see Section 7.2). An object is only stored compressed if it is beneficial, and in this case, compressed object is set to true.

This field is not used for large objects, where subobjects are independently compressed. The compression information is stored in separate subobject descriptors, which will be described in Section 8.5.2.

Class Identifier

In Vagabond, information about a class is stored in a class descriptor object (CDO) (see Section 6.2.1). The class identifier (CID) in an OD is actually the OID of the CDO of the class that the object belongs to.

It is important to note that the class identifier is a temporal property of an object, it can change during the lifetime of the object. With class migration, an object can belong to different object classes at different points in time.

The number of classes, and as a consequence, the number of CDOs, is assumed to be much smaller than the number of objects, so we can manage with a smaller size of the CID than the size of the OID. The class, and the description of it, is global for a database, so there is no need to use a *SGID*. The *CONTID* is also given implicit, we assume the ODs of the CDOs are stored in a separate container. The size of the class identifier is 24 bits, enough to represent over 16 million object classes in one database.

Signature

This optional field contains the object's signature (see Section 7.1.4). The signature field can have variable-length size, if present. Information on signature maintenance and signature size is stored in the CDO.

Inlined Object

In total, 12 bytes in the OD are used to store the physical location and size of an object. If the size of the object is less than 12 bytes, it is better to store it in the OD instead, in the physical location and object size fields. We call this an *inlined object*. Up to 11 bytes is used for the object, while the last byte is used to store the length of the inlined object.

Even though it is also possible to use the signature field for this purpose, that would complicate signature access queries, because the signature would have to be created on the fly every time.

First Version

This bit is set in the OD of the first version of an object. In some temporal operations, this can be utilized to avoid index accesses when we have this OD in main memory.

Migrated to Another Server Group

An object can migrate from the server group where it was created to another server group. In this case, the *migrated to another server group* is set to true, and the server identifier of the new server is stored in the physical location field. If the object is migrated a second time, only the OD on the server on which it was created will be updated. In this way, only one indirection is needed. By caching remote ODs on a server, this will only infrequently require network traffic.

Migrated to Another Container

Similar to migration to another server, an object can be migrated to another container. The new container identifier is stored in the physical location field. In the case the object is migrated a second time, only the OD for the first container will be updated (although it can be wise to update both, to avoid problems with ongoing updates). In this way, only one indirection is needed, similar to the case of server group migration.

8.2 Declustering

One database can be distributed over several server groups (see Section 6.4). In this case, we use one OIDX for each server, and this OIDX indexes the objects stored on this server. In the case of a multi-server system, the OID, which contains the server group identifier, is used to identify which server group to access in order to retrieve an object.

In the case of a server group (see Section 6.4), where data is declustered over the servers in the group, the declustering strategy (for example hashing of OIDs), is used to determine which server in the server group stores that object. In this way, the OIDX is implicitly partitioned.

Figure 8.2 illustrates a distributed system with server groups. In this configuration, we have 4 server groups, and each server group consists of 8 servers. Even though all server groups in this configuration contain the same number of servers, this need not be the case in general. Objects in

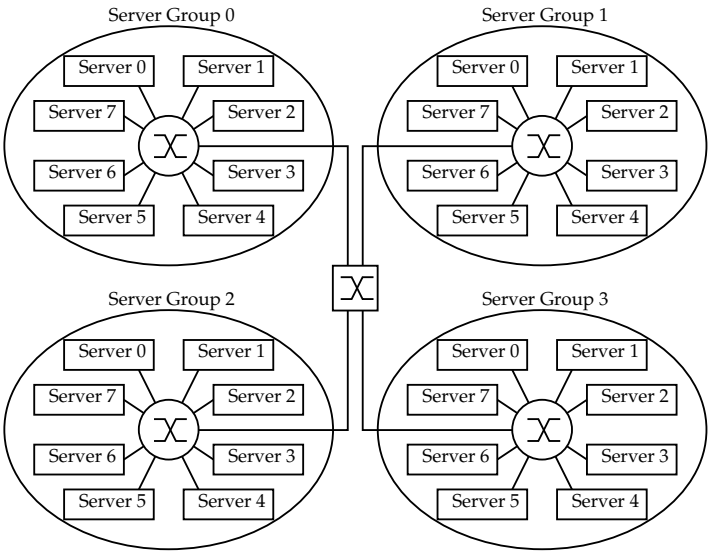


Figure 8.2: Distributed system, with server groups and servers.

one server group are declustered over the servers according to the hash value of their unique serial numbers. If we want to access an object with an OID where $SGID = 2$, $CONTID = 425$, and $USN = 84623$, the server group to access is server group 2, and the actual server in the server group is 84623 MOD 8 = 7. We emphasize that the OIDX of this server *only* indexes the objects that have been created in this server group, i.e., *SGID*$= 2$, *and* have *USN* MOD 8 equal to 7.

When indexing ODs of temporal objects, it is possible that the simple hashing strategy used in this example is not sufficient, and that other declustering schemes can be useful. This will be discussed in Section 11.

8.3 Temporal OID Indexing

In Vagabond, we use one OD for each version of an object (two ODs for migrated objects). The OIDX has to support access to ODs of current as well as historical versions of the objects, and we consider the following requirements as very important for a temporal OIDX in Vagabond:

- Support for temporal data, while still having index performance close to a non-temporal (one-version) DBMS for non-temporal data. Even if the use of other kinds of indexes could give better support for temporal operations, we believe efficient non-temporal operations to be crucial, as they will still be the most frequent operations.
- Efficient object-relational operations. This is expected to be achieved by the use of containers.
- Easy migration of partitions of the index to tertiary storage.

Before we present our temporal OIDX, we take a closer look at some characteristics of OIDs and OID search, and analyze the following four alternatives to OID indexing in a *transaction-time* temporal ODBMS:

1. One index, which indexes ODs of current as well as historical versions of the objects.

2. One index for ODs of current versions, with links to the ODs of historical versions.

3. Nested-tree index, which is one index with *version subindexes*.

4. Two separate indexes, one for ODs of current versions, and one for ODs of historical versions.

8.3.1 Characteristics of OIDs and OID Search

When considering appropriate index structures and operations on these indexes, it is important to keep in mind the properties of an OID:

- The keys in the index, the OIDs, are not uniformly distributed over a domain as keys commonly are assumed to be. If we assume the unique part of an OID to be an integer, new OIDs are in general assigned monotonically increasing values inside a container. In this case, there will never be inserts of new key (OID) values between existing keys (OIDs) in the container. In addition, OIDs will be clustered, in one cluster for each container.

- If an object is deleted, the OID will never be reused.

In a tree-based non-temporal OIDX, new entries will be added append-only. By combining the knowledge of the OIDX properties and using tuned splitting, which will be described in Section 8.4.1, an index space utilization close to 1.0 can be achieved. If container clustering is used, however, inserts in between entries occur, and space utilization will decrease. This can be avoided by using a hierarchy of multi-way tree indexes, as will be shown later.

Without container clustering, index accesses will mostly be for perfect match, there will be no key-range search (in this case a range of OIDs). With container clustering, there will be two search classes: search for perfect match during object-navigation queries, and search for all entries in one container in the case of a container scan. Accessing objects in a container will often result in additional navigational accesses to referenced objects. It is important to remember that there will in general be no correlation between OID and an object-key attribute (if defined), so that an ordinary object key-range search will not imply an OID-range search in the OIDX. If value-based range searches on keys (or other attributes in objects) are frequent, additional secondary indexes should be employed, for example B^+-trees or temporal secondary indexes. In this case, the OIDs (and time in the case of temporal queries) resulted from the key search will be sorted and then used to access the objects by lookups in the OIDX.

In a temporal ODBMS, the existence of object versions increases the complexity. For example, we need to be able to efficiently retrieve ODs of historical as well as current versions of objects, and support time-range search, i.e., retrieve all ODs for objects valid in a certain time interval. To do this, we need a more complex index structure than what is sufficient for a non-temporal ODBMS.

8.3.2 One Index Structure

If only one index is used, we have the choice of using a composite index, which is an extension of the tree-based indexes used in non-temporal ODBMSs, and using one of the general multiversion access methods.

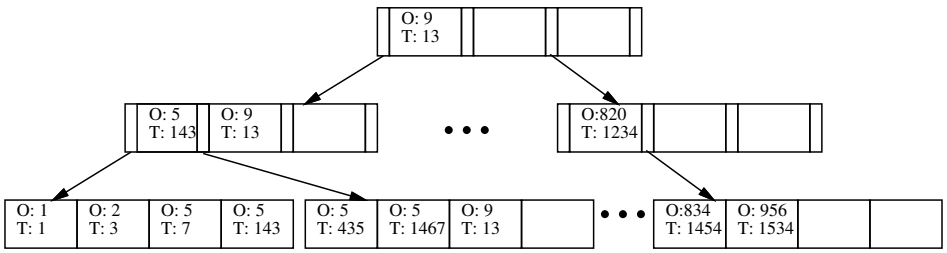


Figure 8.3: One-index structure using the concatenation of OID and commit time, $OID\|TIME$, as the index key.

Composite Index

With this alternative, we use the concatenation of OID and commit time $OID\|TIME$ as the index key, as illustrated in Figure 8.3. By doing this, the ODs of the different versions of an object will be clustered together in the leaf nodes, sorted on commit time. As a result, search for the OD of the current version of a particular object as well as retrieval of ODs for objects created during a particular time interval can be done efficiently.

This is also a useful solution if versioning is used for multiversion concurrency control as well. In that case, both current and *recent* objects will be frequently accessed. It is also possible that many of the future applications of temporal DBMS will access more of the historical data than has been the case until today, something that might make this alternative useful in the future. However, there are two serious drawbacks with this alternative:

1. Even in an index organized in containers, leaf nodes will contain a mix of current and historical ODs. The ODs of current versions are not clustered together, something that makes a scan over the ODs of current versions inefficient.

2. An OIDX is space consuming, a size in the order of 20% of the size of the database itself is not unexpected [62]. In the case of migration of old versions of objects to tertiary storage, it is desirable, and in practice necessary, that parts of the OIDX itself can be migrated as well to avoid the need for large amounts of disk space for the OIDX of the migrated objects. This is difficult when current and historical versions reside on the same leaf pages.

Temporal OID Indexing in POST/C++. One temporal ODBMS using a composite OIDX is the POST/C++ temporal object store [202], which is based on the Texas object store [189]. In POST/C++, objects are indexed with physical OIDs, and a variant of the composite-index structure is used to index historical versions. Because of the use of physical OIDs, a new object is created to hold the previous version when an object is updated. After the previous version has been copied into the new object, the new version is stored where the previous object had previously resided. A positive side effect of doing it this way, is that current and historical objects are separated, and that clustering does not deteriorate.

To be able to access the historical versions, a separate B^+-tree-based history index is used. This index uses the OID of the current-version object, concatenated with time, as the index key. The leaf node entry is the OID of the current version of the object, the time interval where this version was valid, and the OID of the historical version. The location of the historical version is given through the OID in the leaf node.

In a database using physical OIDs, this hybrid index structure is not a bad choice. By doing it this way, current versions will still be clustered together, and having the historical index separated from the current index (in this case no index), makes it easier to migrate historical objects to tertiary storage. The temporal index, on the other hand, can not easily be migrated.

Use of General Multiversion Access Methods

Using one of the general spatial, multidimensional, or multiversion access methods is also an alternative. However, considering the indexing problem simply as spatial/multidimensional indexing with the two dimensions OID and time will not be efficient. An object version is not only valid at a certain time (a point in the multidimensional space), but in a certain time interval (until the next version is created). In addition, a lookup for the current version of an object can be difficult with these index approaches, because time is a constantly expanding dimension. Multiversion access methods are more suitable, but the existing methods have drawbacks. We will here consider three of the most interesting methods: the TSB-tree [132],[3] R-tree [84], and LHAM [143].[4] TSB-trees and R-trees both have efficient support for time-key range search, while LHAM has a very low update cost. However, each of these access methods has drawbacks:

- LHAM is of limited use for OID indexing, because it can have a high lookup cost when the current version is to be searched for. As this will be a very frequently used operation, LHAM is not suitable for our purpose. In addition, which access method to use in the index components is still an issue.
- When indexing ODs, most queries will be OID lookups, and in this case support for key-range search is of little use.
- R-trees are best suited for indexing data that exhibits a high degree of natural clustering in multiple dimensions [178]. This is not the case when indexing ODs, and one of the results is a high degree of overlap in the search regions of the non-leaf nodes. Although a *segment R-tree* can reduce this problem, it will have a higher insert cost [178]. The fact that we do not know the end-time of a new OD further complicates the use of an R-tree.
- Using a TSB-tree or segment R-tree increases the storage space because some entries are resident in more than one node.
- In the TSB-tree, heuristics have to be used to determine when to split by time and when to split by key, and in R-trees, heuristics have to be used to determine bounding rectangles. This makes the performance vulnerable to changing access patterns.

TSB-trees and R-trees have both good support for time-key range search, and make index partitioning possible. However, when indexing ODs, most queries will be OID lookups, and when OID is the key, support for key-range search is of little use. Even if the use of TSB- or R-trees could give better support for temporal operations, we believe efficient non-temporal operations to be crucial, as they will probably still be the most frequently used operations. These multiversion access methods will increase storage space and insert cost considerably, and this contradicts our important goal of supporting temporal data, while still having index performance close to a non-temporal ODB. As

[3]There are also other B-tree-based temporal indexes, including the Write-Once B-tree, the Persistent B-tree and the Multiversion B-tree, but they do not support migration of historical data [178].

[4]See description in Section 5.4.3.

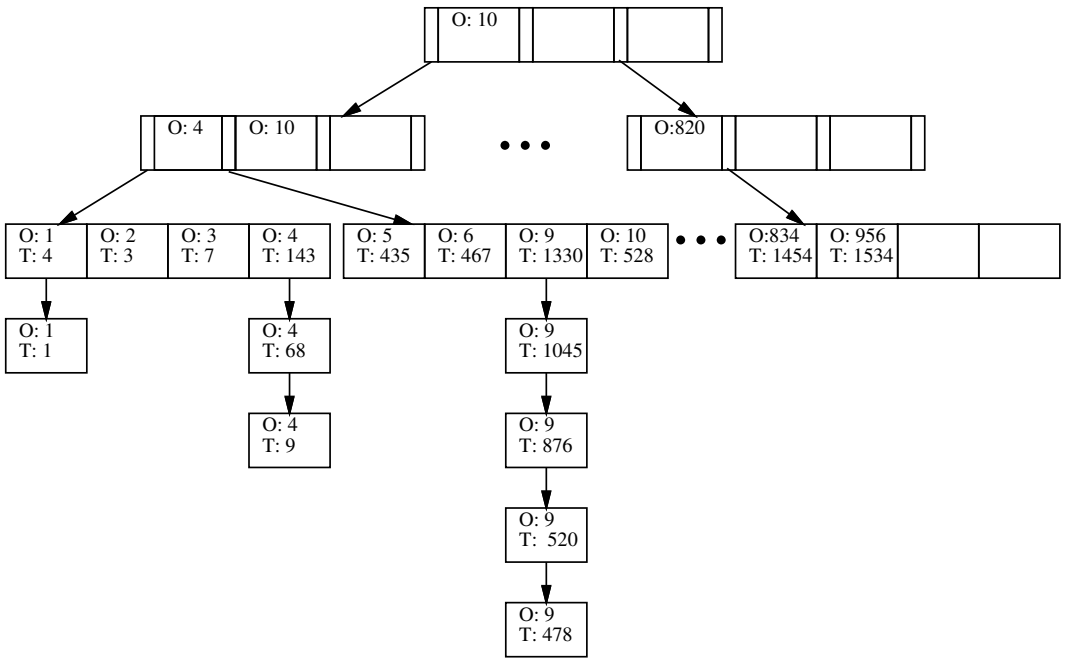


Figure 8.4: One-index structure, with version linking.

a result, we do not consider these as an alternative for the OIDX. However, the general multiversion access methods will typically be used to implement secondary indexes for collections of objects where time/key-range search will be frequent.

8.3.3 Index with Version Linking

To avoid the disadvantages of the composite index, only the ODs of the current versions of the objects are kept in the index. ODs of historical versions, are kept in a linked list. Each OD in the index has a list of ODs of the historical versions, as illustrated in Figure 8.4. The linked-list approach also has serious disadvantages:

1. For all operations on historical versions, the list must be traversed, resulting in extra secondary/tertiary storage accesses. The traversal cost can be reduced by rewriting successor ODs of the chain if they are resident in main memory when a new OD is inserted. If this is done, subsequent traversals will have a lower cost than if the ODs are unclustered.

2. Most important, in a log-only system, the use of a linked list would make cleaning very costly. If we want to move an object, its OD would have to be updated. This implies writing the OD to the log, effectively invalidating the pointer to it from its predecessor in the list. Its predecessor has to be rewritten which again invalidates this OD's predecessor as well. As a result, all the predecessors of the OD of the moved object have to be rewritten. This problem could only be solved by using bidirectional pointers between the ODs. Unfortunately, this would considerably complicate the log writing, because we would have to know the addresses of ODs that are not yet written to the log. This is not a viable solution.

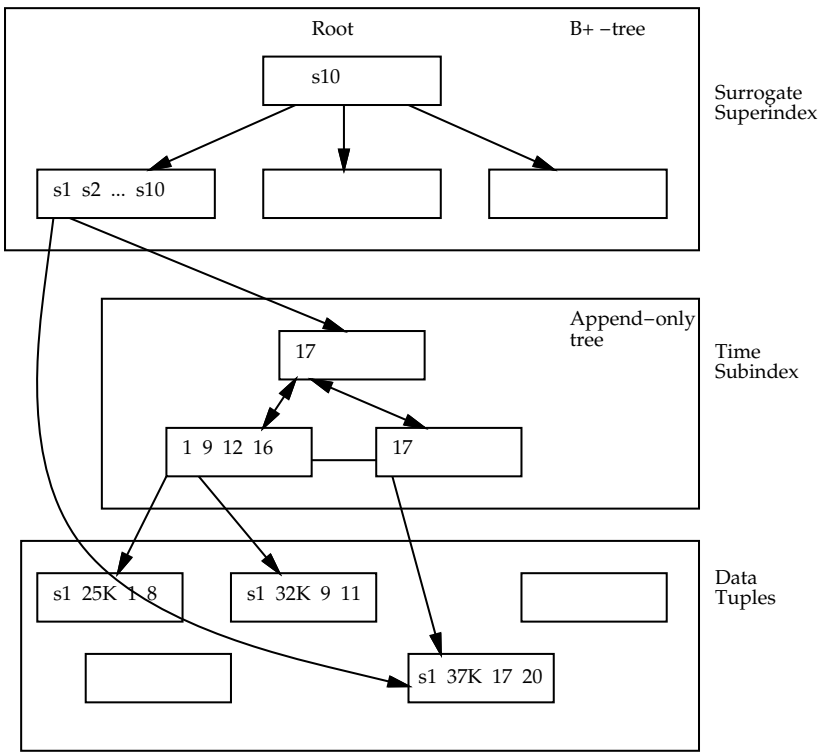


Figure 8.5: Nested Surrogate-Time (ST) indexing [83].

To reduce the access costs for historical versions, it is possible to link the object versions instead of the ODs. This has similarities with the approach used in POSTGRES [195], where a link exists from one version of a tuple to the next (as described in Section 5.4.1). With this approach, the extra access needed to retrieve the object after the OD has been found is avoided. However, we do not necessarily know the size of the next object in the chain, and the cleaning cost increases because we have to retrieve and write objects as well as ODs when an item in the chain is moved.

8.3.4 Nested-Tree Index: Index with Version Subindexes

A better alternative than using lists, is to use a nested-tree index, which indexes current versions in a *superindex*, and historical versions in *subindexes*.

An example of a nested-tree index is the Surrogate-Time (ST) index [83], illustrated in Figure 8.5. The *surrogate superindex* indexes the key values of the tuples, and is implemented as a B^+-tree. Each leaf node entry has a direct pointer to the current data tuple, as well as a pointer to a *time subindex*. The *time subindex* is an append-only multi-way tree which indexes the start time of all (valid-) time intervals. Each entry in the subindex has a pointer to the data tuple that has the timestamp in the key of the entry.

8.3.5 Separate Indexes for Current and Historical Versions

Assuming that most of the queries will be against the current data, it is possible to make read accesses to current version as efficient as possible by using one index for the ODs of current versions of objects,

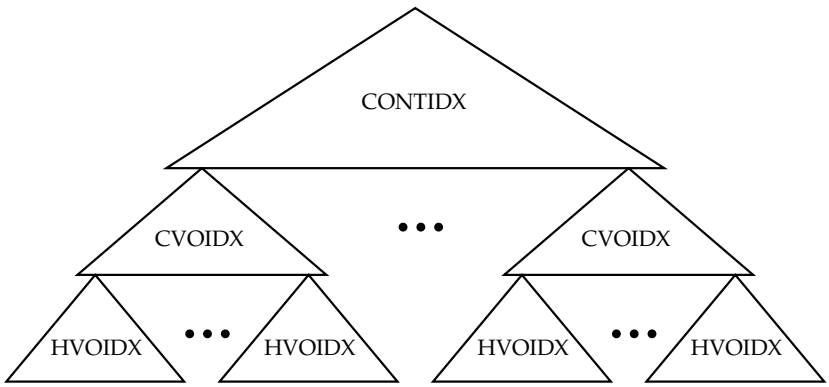


Figure 8.6: The Vagabond temporal OID index.

and a separate index for the ODs of historical versions. The index for the historical data can be organized as either of the three previous index organizations.

The problem with this approach, is that every time a new version is created, we have to update *two* indexes. While this might at first seem to be the case with the previous alternative as well, keep in mind that a subindex tree will in general have a much smaller height than an index indexing all ODs. More important, the size of the index for current versions will be the same as the superindex in the nested-index tree. Also note that even in the context of an in-place update database system where the current version of an object always resides in the same physical location, the current-version index still has to be updated at every object update because the timestamp has changed.

8.4 VTOIDX: The Vagabond Temporal OID Index

In Section 8.3 we described the most important requirements for a temporal OIDX. Based on the analysis of the different OIDX alternatives, we have designed the Vagabond temporal OIDX (VTOIDX). The VTOIDX is a hierarchy of multi-way tree indexes, with three levels, as illustrated in Figure 8.6:

1. Container index (CONTIDX), which indexes the containers in a database.
2. Current-version OIDX (CVOIDX), which indexes all ODs of the current version of the objects in one container.
3. Historical-version subindex (HVOIDX), which indexes ODs of historical versions of objects.

The strict hierarchy in our index might at first look inefficient, as it is likely to result in a higher number of index levels than a solution with one index for all current versions in the database. However, several factors dictate the use of separate indexes for each container:

- By having separate indexes for each container, it is easier to achieve high space utilization, because each subindex is append-only.
- Container migration to tertiary storage is less complex and cheaper.
- With a separate index for each container, it is not necessary to store the *CONTID* for each entry in the nodes (although some of the same effect can be achieved by using prefix compression of

the OID in the index entries). This increases fan-out as well as the number of ODs in the leaf nodes. We expect that as long as the upper levels of the tree are buffered in main memory, the benefits of more ODs in a leaf node outweights the extra cost of the higher number of levels.

We will now describe this indexing approach in more detail, first the physical data organization, and then the operations on the index.

8.4.1 VTOIDX Physical Data Organization

In this section we describe the most important details of the data organization in the VTOIDX. We start with a description of the three indexes in the hierarchy, then describe the use of *subindex caching*, and comment on some additional details of the index trees.

Container Index

There is one CONTIDX for each database, and the CONTIDX indexes the containers in the database. The pointers in the leaf nodes point to a current-version OIDX, one for each container. The entries in the internal nodes as well as the leaf nodes are `(CONTID,pointer)` tuples.

Note that the containers themselves are not versioned, only the contents of the containers. Versioning of the containers would make index management complex and costly, and also occupy more storage. The version of the container (the contents at a given time) is given implicitly by which objects are valid at a certain time.

Current-Version OID Index

There is one CVOIDX for each container, and it indexes the ODs of all the current versions of objects in that container. The CVOIDX and HVOIDX combination is based on the ST index (see Section 8.3.4), and the CVOIDX itself is similar to the surrogate superindex in the ST index.

The entries in the internal nodes in a CVOIDX are `(USN,pointer)` tuples. Because there is a separate index for each container, the *CONTID* is given implicitly. This is also the case for the *SGID*, as each server only indexes the objects that are created on that server.

The leaf nodes of the CVOIDX contain the ODs of the current version of the objects in the container. Similar to the entries in the internal nodes of the CVOIDX, we do not store the *SGID* and *CONTID* part of the OID in the OD, we only store the *USN*. To further increase the number of ODs in a CVOIDX leaf node, prefix compression of the rest of the OD, in particular the *USN*, can be used.

Each CVOIDX leaf node also contain a pointer to the corresponding HVOIDX, which indexes the ODs of historical versions. The CVOIDX leaf node also contains the number of ODs and the smallest and largest *USN* of ODs residing in the actual HVOIDX.

Historical-Version OID Index

The ODs of historical-object versions are stored in HVOIDX subindex trees. It is possible to have one HVOIDX tree for each object, for each CVOIDX node, or shared by several CVOIDX nodes:

- **One HVOIDX for each object:** This is similar to the subindex in the ST index (see Section 8.3.4). The advantages of this alternative are that it is not necessary to store the OID in the ODs in the HVOIDX, and that all inserts to the HVOIDX trees will be append-only. If there are many historical versions of each object, the result will be high space utilization. However,

in many cases, the number of updates to most objects will be low, resulting in few historical versions for each object. As a result, many HVOIDX trees will consist of one almost empty node. The number of CVOIDX leaf nodes will also increase, because they need to store one pointer for each HVOIDX rooted in the leaf node. We expect these problems to outweight the advantages of this alternative. If we can have variable-sized nodes, which will be discussed in Section 8.6.2, one HVOIDX for each object can still be a good alternative.

- **One HVOIDX for each CVOIDX node:** For each leaf node in the CVOIDX, there is a separate HVOIDX subindex tree. Such an HVOIDX contains the ODs of the non-current versions of objects whose ODs resides or have resided in the CVOIDX leaf node. With one HVOIDX for each CVOIDX node we loose the append-only property of the previous alternative, but the space utilization will be better when each object has a small number of historical-object versions. Note that CVOIDX nodes will never be split, so that we never have to "split" a HVOIDX.

- **One HVOIDX shared by several CVOIDX nodes:** When each CVOIDX leaf node has one HVOIDX subtree, it is possible that some HVOIDX subtrees still have only a small number of entries. To optimize space usage, several CVOIDX leaf nodes could share one HVOIDX subtree. However, this would give each HVOIDX root node more than one parent. In a log-only system where the nodes are not updated in-place, the CVOIDX leaf nodes, which are parent nodes of the HVOIDX nodes, would have to be updated as well when the HVOIDX is updated. This would significantly increase the insert cost. We believe the subindex caching introduced below will reduce the need for shared HVOIDXs, and we also assume it is likely that most objects in a container will have the same versioning characteristics.

We want to avoid index roots/nodes with more than one parent, and therefore the only reasonable alternatives are one HVOIDX for each object or one HVOIDX for each CVOIDX node. With fixed-size index nodes the probability of low space utilization is very high for the "one HVOIDX for each object" alternative, and we conclude that one HVOIDX for each CVOIDX node is probably a good tradeoff in HVOIDX sharing.

The entries in the internal nodes of a HVOIDX tree are $(USN\|TIME,\texttt{pointer})$ tuples. The HVOIDX leaf nodes contain ODs only. The concatenation of *USN* and commit time, $USN\|TIME$ is used as the index key during insert and search in an HVOIDX. In this way, we have efficient access to the ODs of a particular object, which will be clustered together, and at the same time we have ODs of current versions in a container clustered.

When indexing non-temporal objects, deleting an object means that the object and its OD can be removed. With temporal objects, however, we need to keep the ODs of an object even when it has been deleted. A tombstone OD is used to represent the delete action, and to store the commit timestamp of the transaction that deleted it. We could either store the tombstone OD in the CVOIDX leaf node where the OD of the current version previously has been stored, or store it in the HVOIDX subtree. To make scans over the current versions of a container as efficient as possible, it is best to store the tombstone OD in the HVOIDX subtree. In this way, the CVOIDX leaf nodes only contain the ODs of the current version of the objects, plus a few ODs of historical versions that have not yet been pushed down to the HVOIDX (subindex caching, see below). Note that in this case, not all OIDs represented in the HVOIDX subtrees are in the CVOIDX leaf nodes, only those of objects that are still alive.

Subindex Caching

When a temporal object is updated, a new OD is created, and the old one is pushed down into an HVOIDX. As a result, the whole path from the leaf in the HVOIDX and up to the root of the CONTIDX must be rewritten (but note that this is done asynchronously and in batch, so that there will in general be more than one OD insertion for each rewritten node, and the average number of written nodes per OD insert is low). In order to reduce the number of nodes to rewrite, and the corresponding installation reads, the ODs of the most recent historical versions are stored in the leaf nodes of the CVOIDX. We call this technique *subindex caching*.

A certain number of slots in the CVOIDX leaf nodes are reserved for ODs of historical versions. In addition, other empty slots can be used. Empty slots will exist when the actual leaf node has not been filled yet (it is the rightmost/most recent leaf node of the CVOIDX), and as a result of object deletions. Only when the CVOIDX leaf node is full, the ODs of the historical versions are "pushed down", in batch, into the HVOIDX tree. This technique will significantly reduce the average update cost.

When we later discuss operations on the VTOIDX, we will consider HVOIDX entries cached in the VTOIDX as a part of the HVOIDX, i.e., when we describe operations on the HVOIDX, this also includes the HVOIDX entries stored in the CVOIDX leaf nodes.

Comments on the Index Trees

Many of the insert operations in the VTOIDX will actually be append operations, i.e., the new entry has a key value larger than any existing key value. In the standard B^+-tree insert algorithm, contents of a split node are distributed over the old and new nodes. If entries are only appended to the index, this would result in a tree with only 50% space utilization. To avoid this, we use *tuned splitting*, a technique also used in the Monotonic B^+-tree [64] for the same purpose. When tuned splitting is used, entries are not distributed evenly over the old and the new node when a node is split, only the new entry is stored in the new node.

In traditional systems, leaf nodes are usually linked together. This can be used to make some of the B^+-tree operations more efficient, and improve concurrency. However, this is not feasible in the case of no-overwrite nodes. Maintaining leaf node links would result in a rewrite of most of the tree each time a leaf node was rewritten.

For all the trees in the VTOIDX, we employ a *no merge/remove on empty* strategy. With this strategy, nodes are not merged when the space utilization in the node gets under a certain threshold because of deleted entries. Only when a node is empty, it will be removed. This is commonly used in B^+-tree-implementations, because 1) merging is costly, 2) in general, there is a certain risk that a split might happen again in the near future, and 3) in practice, this strategy does not result in low space utilization [77]. In the CONTIDX and CVOIDX we know that *CONTID*s and *USN*s will not be reused, and that we will have no inserts, only append operations. Delete operations can still be too costly, especially because they will involve subtrees as well. For this reason, we use the no merge/remove on empty strategy in these indexes as well, and instead rely on background reorganization of the indexes to compact index page with low space utilization.

In the HVOIDX nodes, binary trees are used *inside* the nodes to reduce CPU cost. If entries in a node were not organized in an efficient access structure, we would need to either use sequential search as the access method, or keep the entries sorted, reordering the contents of the node each time we did an insert or delete. If we assume less than 64 K entries in a node, which is the maximum number of entries that can be indexed using two bytes, this gives an overhead of approximately 4 bytes for

each entry. Using binary trees inside the nodes in the CONTIDX and CVOIDX trees is not necessary, because there will be no entries inserted into the nodes in these trees. All entries added to the nodes in these trees will be appended. In this way, the entries are always sorted, and binary search can be used when searching. When an entry is deleted, the slot of the deleted entry can be marked with a null value, and we maintain a counter of the number of deleted values, so that we know when there are no entries left.

8.4.2 Operations on the Vagabond Temporal OIDX

In this section we describe the most important operations on the VTOIDX.

Creating or Deleting Containers

Creating a new container is done by inserting a new entry into the CONTIDX. The value of a new *CONTID* will always be larger than existing *CONTID*s, so this will actually be an append operation, and tuned splitting (see Section 8.4.1) is used to achieve high space utilization.

Physically deleting a container is done by deleting the container entry in the CONTIDX. This operation should be done after the corresponding CVOIDX and HVOIDX indexes have been deleted. If the CVOIDX and HVOIDX indexes are not deleted, the live byte counter (see Section 5.1.2) of the segments they are stored in will not be decremented, and they will still occupy space.

While physically deleting a container is easy, the consequences can be more troublesome. In the case of deleting a whole database, there are no problems, because there should be no accesses to objects in a non-existant database at a later time. Deleting a container, on the other hand, is can cause problems. There can be references to the objects in the deleted container from other objects. If the current version of an object references an object in a deleted container, that is probably an error, but previous versions of objects might reference objects in the deleted container as well. This gives us three alternatives. Which alternative to choose, should be up to the database administrator:

1. Require the application code to do some kind of exception handling when a temporal query tries to access a deleted container.

2. Keep the CVOIDX and associated HVOIDXs in the system, but flag all update attempts as errors.

3. Let the database system verify that there are not references to objects in a container before it can be deleted. This can be a very costly operation, but it will often be possible to know which objects (which classes or containers) that have to be checked, so that the number of objects to check can be low enough to make this alternative interesting.

Create New Objects

When a new object is created, the application that creates the object decides which container the object (and the OD) should belong to, and a new OID is allocated.

If the transaction commits, the OD of the new object is inserted into the VTOIDX. This is done by first retrieving the actual CVOIDX root node in the same way as when searching for an object. If there is free space in the rightmost leaf node, the OD of the new object is inserted there. If not, a new CVOIDX leaf node is allocated, and the new OD is stored there. If there is overflow in the parent node, a new node is allocated at that level as well. This applies recursively to the top of the index tree.

As with all tree-based indexes that are stored in a log, all the ancestors of an updated index node have to be updated as well. This also includes nodes in higher-level indexes.

Search for Current Object Version

Search for the OD of the current version is done by first using the *CONTID*, which is a part of the OID, to do a lookup in the CONTIDX to get a pointer to the CVOIDX where the OD resides. When the CVOIDX root node has been retrieved, the *USN* of the OID is used to search the CVOIDX, and if the object with the searched OID exists and is valid, its OD will be found in a CVOIDX leaf node. For both the search in the CONTIDX and the CVOIDX, the standard multi-way tree search algorithm is used.

Update Temporal Object

When a temporal object is updated, the old OD is replaced with the OD of the new version, and the OD of the previous current version is inserted into the HVOIDX subindex.

The first step is to find the CVOIDX leaf node where the current version of the OD is stored, and replace this OD. When the OD of the previous version is inserted into the HVOIDX, the concatenation of *USN* and commit time, $USN||TIME$, is used as the HVOIDX index key. Note that in this case, we also have inserts into the tree, and not only append operations. Therefore, we use the standard B^+-tree insert algorithm, without employing tuned split. To reduce the average update cost, we also employ subindex caching as described in Section 8.4.1.

Update Non-Temporal Object

The operations on the OIDX when updating non-temporal objects are made more complicated because of the existence of delta objects. Even though we in general do not keep previous versions of non-temporal objects, we have to do this when writing delta objects. Previous versions back to the last complete version have to be kept, and this has to be handled when maintaining the OIDX in the case of object updates:

1. If the new version is a delta object, the previous version(s) of the ODs of the object have to be kept. The update is done exactly in the same way as when updating a temporal object.

2. If the new version is a complete object, i.e., not a delta object, and the previous version was a delta object, there are two or more ODs of this object: one OD for the most recent complete object, and one OD for each subsequent delta version. All these ODs should be removed when we write a new complete object.

3. If the new version is a complete object, and the previous version was also a complete object, we know that there is only one OD for this object in the OIDX. In this case, the old OD is simply replaced with the new one, and the leaf node and its ancestors are updated.

Delete Object

In the case of a non-temporal object, the OD is simply removed from the actual CVOIDX leaf node where it resides, and the leaf node and its ancestors are updated. If the object was a delta object, there is more than one OD for this object, and all of them have to be removed.

In the case of a temporal object, the current OD and an additional tombstone OD are inserted into the HVOIDX subindex (the tombstone OD is an OD where physical location is NULL, and the timestamp is the commit time of the transaction that deleted it, as described in Section 8.1.2). The current OD is deleted from the CVOIDX, and the affected HVOIDX and CVOIDX nodes and their ancestors are updated.

As mentioned previously, we use a *no merge/remove on empty* strategy, nodes are not merged when the space utilization in the nodes gets under a certain threshold. Only when a node is empty, will it be removed. When a CVOIDX node is removed, the entries in the HVOIDX subtree are inserted into the HVOIDX of one of its two neighbor nodes. The *USN* range and the HVOIDX counter in the CVOIDX node are updated to reflect the change.

Vacuuming

An old version of an object can not be updated or deleted. However, old versions can be vacuumed (see Section 4.5). During vacuuming, ODs residing in the HVOIDX can be deleted. This is done according to the standard B^+-tree delete algorithm.

Search for Object Version Valid at Time t_i

First, a search is conducted to find the CVOIDX leaf node where the OD of the current version of the object resides. If the timestamp in this OD is less than t_i, this OD is the result of the search. If not, the HVOIDX is searched to find the OD of this object that have the *largest timestamp less than* t_i. Note that the ODs of deleted objects only reside in the HVOIDX (except those that are subindex cached). Thus, even if an OD with the actual OID is not found in the CVOIDX leaf node, we still have to search the HVOIDX.

Search for All Versions of an Object

This operation is done by first retrieving the CVOIDX leaf node where the OD of the current version of the object resides (or where it has previously resided, if the object is deleted), and then retrieving the ODs of all versions of this object from the corresponding HVOIDX.

Search for Start or End Time of an Object

To find the time an object was created, a lookup is done to find the OD of the first version of the object. Similarly, to find the end time of an object, a lookup is done to find the OD of the last version of the object.

Search for Current Version of all Objects in a Container

This is the traditional scan operation. In the VTOIDX, this is done in the same way as in a traditional B^+-tree, by returning all ODs from all leaf nodes in the CVOIDX of a container.

Search for Objects in a Container Valid at Time t_i

In this operation, all the CVOIDX leaf nodes of a container have to be searched for matching ODs. Because deleted objects are not represented in the leaf node, *all* HVOIDX subindexes have to be searched as well, because they may have ODs of deleted objects valid at time t_i. The only case where the search in the HVOIDX subindex can be avoided is:

- If the *USN*s of the ODs in the CVOIDX leaf node represent a contiguous range. In that case, we know there will be no ODs of deleted objects in the HVOIDX subindex.
- *And* all ODs in the CVOIDX leaf node have a timestamp older than t_i.

If this type of query is expected to be frequent, it would be beneficial to keep the tombstone ODs of deleted objects in the CVOIDX leaf nodes. If this is done, we could avoid further searches in the HVOIDX if all objects represented by the particular CVOIDX leaf node was deleted before time t_i. In that case, we know that they could not be valid at time t_i. As we do not know for sure the frequencies of different query types in future systems, it is difficult to say if this kind of query will be frequent enough to justify keeping tombstones of deleted objects in the CVOIDX leaf nodes. Anyway, this is the kind of query where an additional secondary temporal index would be useful, and it is possible that relying on secondary indexes would be better in this case, as it would not affect current-version OD scans.

Update all Objects in a Container

Only objects that are still valid can be updated. Each object update creates a new OD to be inserted into the VTOIDX. Although this operation could be performed by repeated lookups and updates, the operation should be done in batch. When inserting ODs into the container in this way, the OIDX operations will have a low average cost.

Migration to Tertiary Storage

Any subtree of the VTOIDX can be migrated to tertiary storage. The subtree can be part of a HVOIDX, part of an CVOIDX and the HVOIDXes under it, or even part of the CONTIDX and the CVOIDXes and HVOIDXes under it. All nodes in all levels of the VTOIDX are addressed by a 64 bit physical location address. The physical location is a location in the log, on the data volume (see Section 6.1.5).[5]

Lookups in an index stored on tertiary storage will be costly compared with lookups in an index stored on secondary storage. This is especially the case for single OD lookups. The cost of scan operations is relatively cheaper, especially in the case of a large subtree. When accessing tertiary storage, the main cost is usually the seek time. The data transfer itself can usually be done with a relatively high bandwidth. When a subtree is migrated to tertiary storage, it should be written in a way that make scan operations on the subtree as inexpensive as possible.

If the contents of an index node stored on tertiary storage are updated, this necessitates a rewrite of the branch it belongs to. This may or may not be done directly to tertiary storage.

8.5 Large Objects

As described in Section 6.2, all objects smaller than the *subobject threshold*, are written as one contiguous object in the log, while objects larger than this threshold are segmented into *subobjects*. The subobject threshold can be different for different object classes.

There are several ways to manage the subobjects, but all should satisfy one important requirement: *storage efficient versioning*, i.e., when a large object is updated, as few subobjects as possible should

[5]Note that also tertiary storage is considered part of the log.

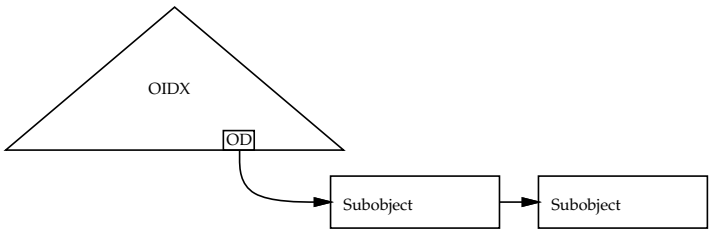


Figure 8.7: Storage of a large object when using the linked-list approach.

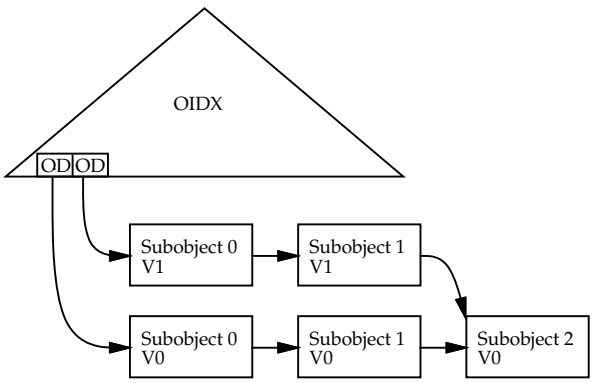


Figure 8.8: Versioned large object, when using the linked-list approach. V0 denotes the first version, V1 denotes the second version.

be rewritten. In this section, we discuss two alternative methods to manage subobjects: the *linked-list* approach, and the *subobject-index* approach.

8.5.1 Linked List

In the linked-list approach, subobjects are stored as a linked list, as illustrated in Figure 8.7. If this approach is used, the physical location in the OD of the large object is the location of the first subobject. Each subobject contains a pointer (physical address) to the next subobject in the list.

As described previously in this chapter, we use a separate OD in the OIDX for each version of an object. Using a linked-list approach, each of the ODs of an object points to a linked list representing the actual version. Figure 8.8 illustrates the two versions of a large object that exist after object creation and one update. Version 0 is the initial large object, resulting from the transaction that created the large object. The object is segmented into three subobjects. Later, another transaction modifies subobject 1 (the second subobject) of the large object. A new linked list is created for this version, and a new OD is inserted into the OIDX. As illustrated in the figure, we have to duplicate the predecessors of the updated subobject. This is not ideal, but necessary for two reasons:

1. In a list, there is only one link out from each subobject, so that we have to duplicate the predecessors of the updated subobject.

2. We can not have bidirectional links when using the log-only approach, and as a result, all predecessors of the modified subobject need to be rewritten.

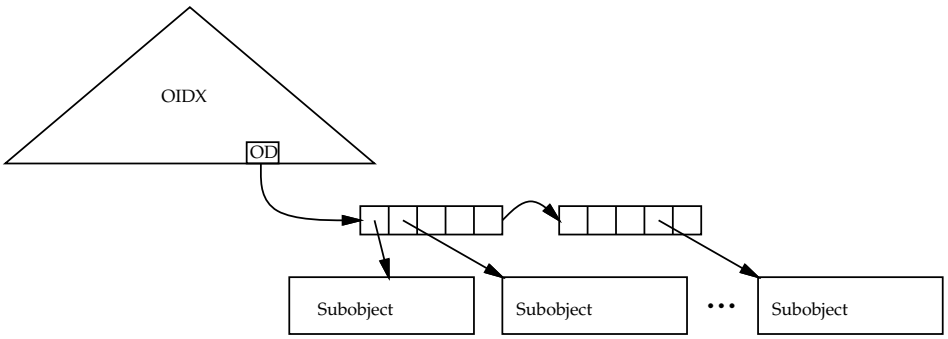


Figure 8.9: Storage of a large object when using a linked list of pointer arrays.

The advantage of the linked-list approach is that if the number of subobjects in a large object is small, accesses can be less costly than by using an additional subobject index. However, there are several problems with this approach:

- Accesses to large objects are efficient if accesses are mostly sequential scans through the objects, starting with reading the first subobject, which is linked from the OIDX. If accesses do not follow this pattern, it will be necessary to read several subobjects from secondary/tertiary storage in order to retrieve the subobject(s) to be accessed.[6]

- Updates to parts of an object are costly. We have to first traverse the list to find the subobject to be updated. When the new subobject has been written, the pointer in its predecessor will be invalid, so we have to rewrite this and all the other predecessors.

- Cleaning can be very costly. When moving subobjects during cleaning, we have the same situation as when doing updates, it is necessary to rewrite all predecessors of the moved subobject.

- The versioning cost is high, because all predecessors of an updated subobject have to be stored.

An obvious improvement to the linked-list approach is to store the pointers to the subobjects in a pointer array, which itself can be linked. This is illustrated in Figure 8.9. However, as the number of subobjects and pointer "subarrays" grows, many of the problems discussed above appear again. The solution is a generalization of the pointer array, a *subobject index*.

8.5.2 Subobject Index

The approach used to manage subobjects in Vagabond, is *subobject indexes*, illustrated in Figure 8.10. Each large object has a separate subobject index which is used to access the subobjects. This subobject index also takes care of versioning, without the problems of the linked-list approach. The subobject indexes used in Vagabond is heavily based on the EXODUS large storage objects [38], and we will now give an overview of EXODUS large objects, before we discuss some extensions that we have included in the Vagabond subobject indexes.

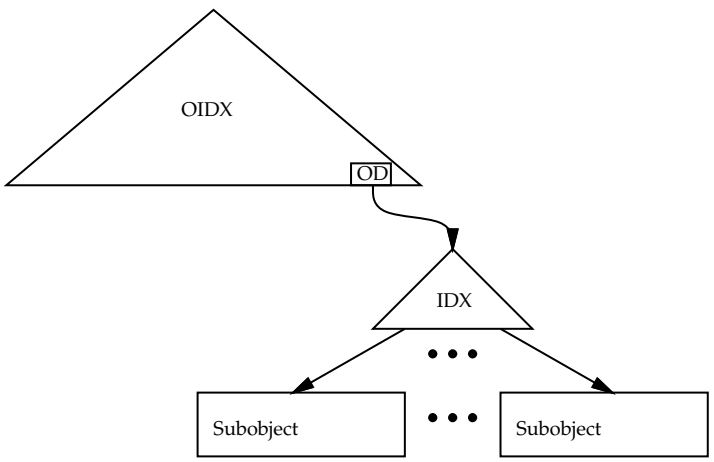


Figure 8.10: Storage of a large object when using the subobject-index approach.

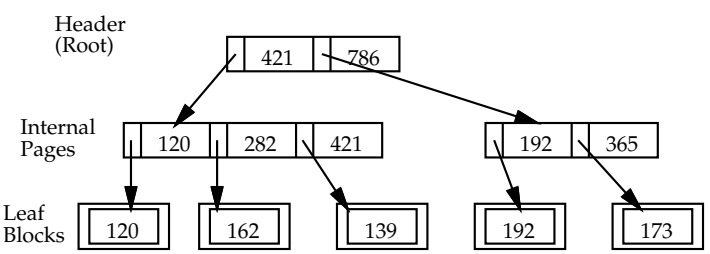


Figure 8.11: Large object in EXODUS [38].

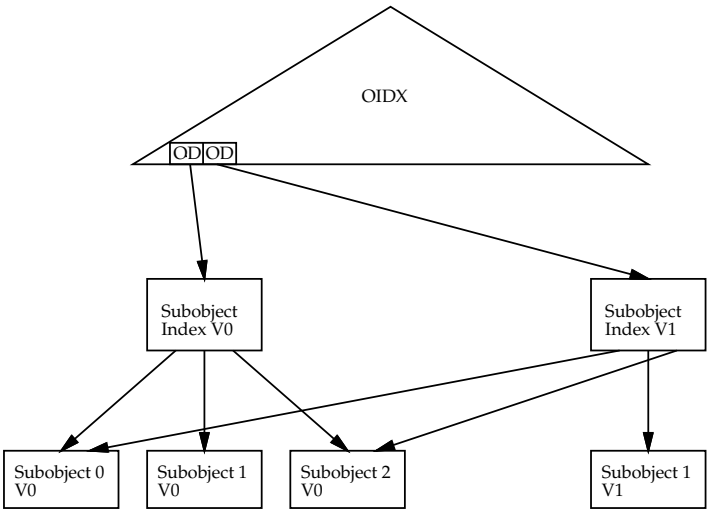


Figure 8.12: Versioned large object. V0 denotes the first version, V1 denotes the second version.

EXODUS Large Objects

Large objects in EXODUS can be viewed as an array of bytes, where it is possible to insert bytes and append bytes to the array, and retrieve a range of bytes. The large object is indexed by a B^+-like tree, where entries in the root and internal nodes are `(count,pointer)` tuples. The count value is the maximum byte number (position in the large object) stored in the subtree rooted from the pointer in this entry. This is illustrated in Figure 8.11. In the figure, the numbers in the internal nodes are byte counts, the number in the leaf nodes are the number of bytes in the actual leaf nodes. In the leftmost entry at the root node, 421 is stored. This is the number of bytes in the subtree rooted at this entry, i.e., $120 + 162 + 139$. The advantage of this organization, is that it is easy to insert into the large object without updating the whole tree, only an upward propagation is needed. A detailed description of the EXODUS large object management algorithms is given in [38].

The EXODUS storage system also supports versioning, as illustrated in Figure 8.12. On top of the figure is the OIDX, which has separate ODs for each version of an object (in EXODUS, physical OIDs are used, so there is not actually an OIDX). To the left we have the initial version of an indexed large object. When parts of the object are updated, only the updated subobjects are stored. The index structure for the new version points to the previous version of the subobjects for those parts that have not been modified. This versioning also applies to the large object indexes themselves. If the indexes have more than one level, only modified parts of the index are rewritten, as illustrated in Figure 8.13. The result is minimal duplication of subobjects, efficient update of the subobject index, and efficient access to parts of the large objects.

Vagabond Large Objects

To make large objects flexible and to support efficient implementation of the special objects (see Section 6.2.4), we have extended the EXODUS large storage object. In the internal large object index nodes, we use `(NavigDesc,pointer)` tuples instead of `(count,pointer)` tuples, where the

[6] Actually, we only need to retrieve the header of each subobject, to get the pointer to the next subobject. However, each subobject read will be a random access read, giving a high seek time.

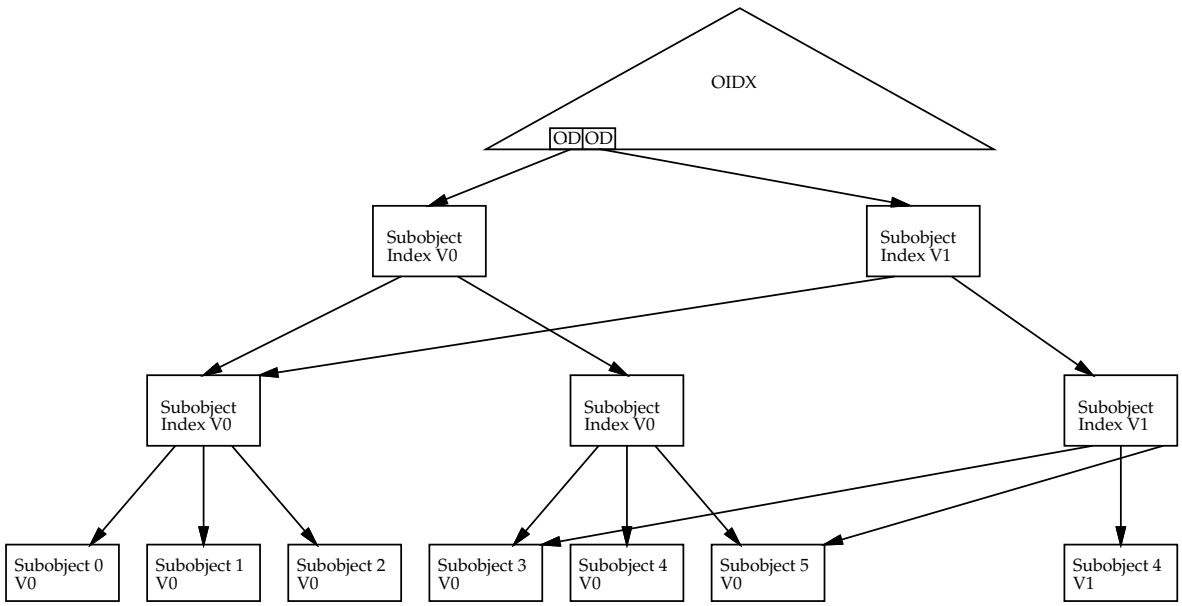


Figure 8.13: Versioned large object in the case of a multi-level subobject index. V0 denotes the first version, V1 denotes the second version.

`NavigDesc` (Navigational Descriptor) is a variable-length structure. In this way, a Vagabond large object is a generalization of the EXODUS large storage objects. EXODUS large storage objects can be realized by using a counter as the `NavigDesc`, this is the default case.

More complex indexes and other special objects can be implemented as more general Vagabond large objects by using more complex `NavigDescs`. For example, R-trees can be realized by using rectangle descriptors in the `NavigDesc` data structure, a `(timestamp,key)` tuple can be used for temporal indexes etc. The size of the `NavigDesc` structure is stored in the CDO (see Section 6.2.1). Chapter 10 discusses in more detail some aspects of Vagabond large objects.

The entries in the leaf nodes of the subobject indexes are subobject descriptors (SODs). The SODs are also stored together with the subobject in the segments.

The contents of an SOD are summarized in Figure 8.14, together with the size of the individual

Field	Size (bits)
OID:	
CONTID	32 (Only present when outside the subobject index)
USN	64 (Only present when outside the subobject index)
Physical location	64
Write timestamp	64
Subobject size	14
Delta subobject?	1
Compressed object?	1

Figure 8.14: Contents and size of fields in the subobject descriptor (SOD).

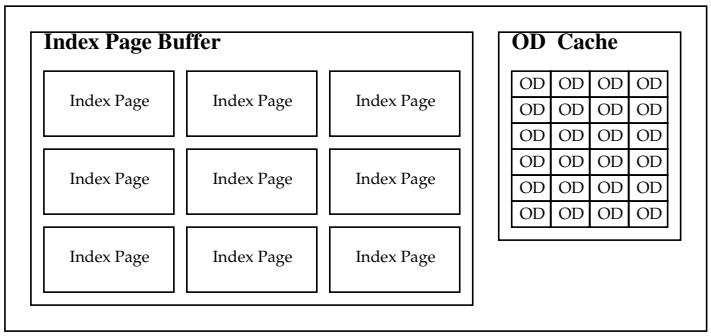


Figure 8.15: Index page buffer and OD Cache.

fields. The meaning of each field is similar to the same fields in the OD (see Section 8.1.2), with two exceptions:

1. The write timestamp is the time when the subobject was first written to a segment. This time is in general before the commit time of the actual transaction. When a subobject is moved because of cleaning, the write timestamp remains the same.

2. The subobject size is the number of secondary/tertiary storage blocks the subobject occupies. The size of storage blocks will typically by 512 bytes or a multiple of 1024 bytes.

8.6 Reducing the OIDX Access Costs

OID indexing in a temporal ODBMS can easily become a bottleneck. A new OD has to be added to the OIDX every time an object is updated, which contrasts to a non-temporal ODBMS, where the OIDX is only updated when an object is created, deleted, or moved/reclustered. Adding an entry to the OIDX when an object is created is relatively cheap, because most ODs of new objects can be inserted into the OIDX by efficient append-only operations. Inserting an OD into the OIDX, which is done when an object is updated, is more costly. The entries in an OIDX have in general a low locality, which often makes it necessary to update one index leaf node for each entry to be inserted.

In a log-only system, the cost is also increased due to the log-only strategy itself. When an index node is updated, it will be written to a new location. The pointer in its parent node becomes invalid, and the parent node needs to be updated. This cascades up to the root.

Several strategies can be employed to minimize the OIDX bottleneck. We will in this section describe how the OIDX access costs can be reduced by using a separate *OD cache*, and how to make use of the possibility of using variable-length index nodes. In Chapter 9 we introduce the *persistent cache*, which will further reduce the OIDX access cost.

8.6.1 OD Cache

Traditionally, the most recently used *index pages* have been kept in a buffer pool to make OID mapping efficient. However, if the index entries have low locality, which is often the case in an OIDX, only a little part of the information on the pages in the buffer is really of interest. To better utilize the memory, it is possible to keep the most recently used *ODs* in an OD cache (see Figure 8.15), like

it is done in the Shore ODBMS [136]. By the use of an OD cache, the OIDX *lookup cost* can be considerably reduced.

When we want to insert an OD into the OIDX, we first have to traverse the OIDX to retrieve the leaf node where the entry is to be stored. The retrieval of index nodes during the traversal of the tree will be random reads. Even though the upper levels of the tree will be in the index-node buffer, this traversal will be costly. To reduce the installation-read cost, it is possible to do some extensions to the OD cache technique. In addition to using it as a lookup cache, we also update and insert entries into it. When a transaction commits, the new ODs are written to the log, so that the index nodes can be updated later. This will reduce the cost by allowing ODs of frequently used non-temporal objects to be updated several times before they need to be written back to the OIDX, and increases the probability that we will have more than one dirty OD for each OIDX node to be updated. Also, by not updating the OIDX immediately, some of the index-leaf nodes where the ODs belong in, will be read into memory because of requests for other ODs residing on these nodes, and this way, we avoid some of the extra installation reads.

8.6.2 Variable-Sized Index Nodes

Using variable-sized index nodes can significantly reduce the index-node write cost. The savings in read cost are less important, because most index nodes are read in random reads, where seek time dominates the read cost.

We will now study two alternatives for variable-sized index nodes. First, we study the case where the index node size is variable, but fixed within a container, and then we study the case where *all* index nodes are variable sized, even within a container. The first alternative, where the size is fixed within a container, can also be employed in a system using in-place updating, but variable-sized index node size within a container is only efficient when we are not constrained to in-place updating.

Fixed Within A Container

In a container where accesses have low locality, most lookups and updates are applied to only a single entry in each index node. For such containers, the write cost can be considerably reduced if relatively small index nodes are used. Reading index nodes can also be less costly, but as said previously, seek time dominates, so the reduced read cost is usually not significant.

Smaller index nodes can increase the number of levels in the tree. If not enough of the upper level nodes fit in the index-node buffer, this can result in *increased* access costs. This means that it is important to use suitable node sizes. With low locality in accesses, which is typical for containers that are mostly accessed as a part of navigational access, we have found node sizes of 2 KB to be optimal for typical access patterns (see Appendix D). Other containers will have high locality. This is typically containers which are accessed often during scan queries. For these containers, larger index nodes will be beneficial.

Variable-sized index nodes, with a fixed-size within a container, are easy to integrate into a system (but the index-node buffer has to support different node sizes). The size of the index nodes in a container can be stored in the CONTIDX, together with the pointer to the container.

Variable Index-Node Size

We have now discussed how using different index node sizes for different containers can be employed to reduce the OIDX access cost. A more radical approach is to let *all index nodes be of variable size*.

This can be useful for many index types. One example is a B^+-tree, where a node is on average only 69% full.[7] The access cost could be significantly reduced if we only had to read and write the part of a node that actually contained data. In the VTOIDX, this is especially relevant for the CVOIDX and HVOIDX. Even though the CVOIDX is append-only, and the nodes in most cases will be full when they are written, the number of entries in the nodes will decline as objects are deleted. The HVOIDX behaves similarly to an ordinary B^+-tree, with corresponding space utilization.

Combining variable-sized index nodes with subindex caching (see Section 8.4.1) can be very beneficial. Index nodes can grow up to a certain size before the ODs of historical entries are pushed down to the HVOIDX in a batch operation.

It is obvious that the index write cost can be reduced with this approach, but to benefit from variable-sized index nodes when reading the nodes, we need to use one of the following strategies:

- Store the size of the index node in its parent node in the index tree, so that we know the size of the block we want to read from secondary or tertiary storage.
- Define a maximum size of the index nodes of a container, and always read a block of this maximum size, as is done in a traditional system. The size of the node is stored in the header of the nodes, so that when we read it, we know how much of the block that we retrieved from secondary or tertiary storage is actually a part of this node.

If the first alternative is used, the size of internal nodes can increase. However, if the variable-sized nodes can only have a fixed number of sizes, only a few bits would be needed to represent the node size. The fact that the parent node containing the size of the index node has to be written when we write an index node, incurs no extra cost in a log-only system, because it has to be written anyway (we always have to write index nodes bottom up in a log-only system).

We consider the second alternative as the most appropriate. This strategy is "cheaper", because we do not need to store the index node size in the parent node. Seek time usually dominates, so that we do not loose much of the benefits even if we always read a block with a size of the maximum index node size for the actual container.

8.7 Log-Based vs. In-Place Updated OIDX

The VTOIDX as described in this chapter can be used in traditional in-place update-based systems, as well as log-only systems. If used in a traditional system, the OIDX will naturally be implemented as a separate structure, updated in-place, with WAL. In the case of a log-only system, we have two alternatives:

1. Log-Based OIDX: The OIDX can be implemented according to the log-only approach, where objects as well as the OIDX nodes are stored interleaved in the log.
2. In-place updated OIDX: The OIDX can be implemented as a separate structure, which is updated in-place. Objects and the index-logging information are still stored in the log. The index could be updated lazily, in batch, and WAL could be used for recovery purposes. The VTOIDX is based on traditional multi-way trees, so the relevant logging and recovery techniques are well known. With this approach, the cost of cascading node updates (bottom up updating) is avoided.

[7] In practice, when using the no merge/remove on empty strategy, the space utilization will be less than 69%, a typical value is 50% [77].

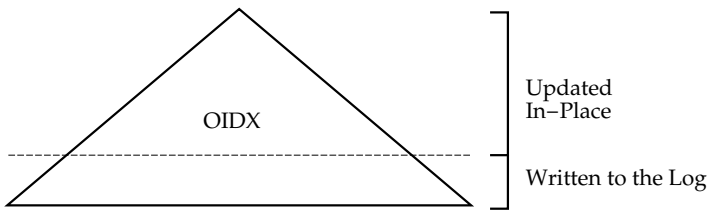


Figure 8.16: Hybrid OIDX where the upper levels of the OIDX are kept in a separate OIDX, and updated in-place, while the lower levels of the OIDX are stored in the log.

It is possible that in databases for some application areas, very few objects will be deleted. If we did not remove the ODs of deleted objects in the CVOIDX, the CVOIDX could actually be implemented as an array instead of using a multi-way tree (this approach could only be used for a separate OIDX, if using a log-based OIDX, a separate search tree would be needed for the nodes). This would eliminate the need for the upper levels of the CVOIDX and the need for storing the OID in the OD in the CVOIDX. Calculating the node and the position of a particular OD in a node would be easy, so that we would not need an access structure inside the node. This approach would cut the OIDX size considerable and would also make OD lookups more efficient. However, the ODs of deleted objects would occupy space, and the practical use of this approach is very dependent of application and access pattern.

Using a separate OIDX might at first look like an interesting solution, because we do not have to write all nodes on the path from the updated node to the root node. However, there are some details that make this solution less appealing than it might be at a first glance:

- A system might crash during an in-place update of an index node, and the node is only partially written to secondary or tertiary storage. A situation like this can be handled by the use of a cold log or backup together with the warm log, or by always writing the before image of the index node (or a logical logging record) to the log before it is updated. Using a cold log or backup means a long recovery time, while writing before images of the index nodes generates large amounts of log.
- To keep down the recovery time, the time between two checkpoints can not be too long. During this time, only a fraction of the index will be accessed between each checkpoint. As a result, we would need to do a random read of the before image of an index node *as well as a random write* to update the index node. This is very costly.[8]
- Using WAL and in-place updating necessitates more complex recovery handling, and we loose some of the benefits of the log-only approach: to keep recovery simple and fast.

An interesting hybrid solution is an OIDX where the upper levels of the OIDX are kept in a separate OIDX, and updated in-place, while the lower levels of the OIDX are stored in the log (see Figure 8.16). In this way, the OIDX nodes that are most frequently accessed are updated in-place, and the less frequently accessed nodes are written to the log.

In the following chapters, we only consider the log-based approach, but in the performance analysis later in this thesis we leave this issue open.

[8]This applies to traditional ODBMSs as well, but in the case of non-temporal ODBMSs, the index does not need to be modified each time an object is modified, and therefore does not represent the same problem.

8.8 Object References and Remote Objects

Objects often contain references to other objects. Using the whole OID, which includes the *SGID*, the *CONTID*, as well as the *USN*, implies a large overhead. To reduce the overhead, it is possible to follow the approach used in Shore [136]. In Shore, one bit of the object reference is used to tell if the reference is to a local object, residing on the same volume, or a remote object, residing on another volume, possibly on another server. If it is a local object, the reference is the serial number of that object. If it is a reference to a remote object, an additional *remote OID index* (ROIDX) is used to do the mapping from the reference to the complete remote OID, which includes a server group identifier and a serial number (the serial number in the remote OID is in general different from the reference). Most references are local, and the Shore approach reduces storage consumption significantly. The extra lookup cost is in general marginal to the total cost of accessing a remote object.

In Vagabond, the same approach as in Shore could be used. The ROIDX would be realized as an ordinary secondary index, which is stored as an object similar to the other secondary indexes in Vagabond. The OD of the ROIDX object, and in most cases the top levels of the index as well, will usually be resident in the buffer, so the lookup cost will be small.

In Vagabond, the ROIDX can also be realized in another way, by using forwarding. In this case, all references in objects are implicitly references to objects stored on the same server group. If the object is stored on another server group, the "migrated to another server group" bit in the OD is used to signal that this is a remote object. The physical location and timestamp fields in the OD is used to store the *SGID*, *CONTID*, and *USN* of the remote object. Note that the reason that this is not an alternative solution in Shore, is that the ODs of a non-temporal ODBMS are not large enough to store the forwarding information.

8.9 Tertiary Storage Indexing

Extending a system to handle tertiary storage in a transparent *and* scalable way is not trivial. As long as the OIDX fits on secondary storage, the issue can be solved by reserving parts of the logical address space for tertiary storage. However, as long as the objects are not very large, this only works for databases that are only a few times larger than the available secondary storage space, because the OIDX itself will occupy large amounts of space. Two suitable approaches for our purpose are:

- An approach based on the log-structured merge-tree.
- Container migration.

8.9.1 Log-Structured Merge-Tree

One way to solve the tertiary storage problem, is to use techniques from the log-structured merge-tree (LSM) [169], described in Section 5.4.3.

The LSM can be used to realize indexes and databases that span several levels in the storage hierarchy, including tertiary storage. If we treat the OIDX on disk as C_0, we can migrate an OD to component C_1 on tertiary storage together with the object (note that an OD should not be migrated unless the object is migrated as well, because as long as the valid version of an object resides in a segment, the OD can be needed at cleaning time). In general indexing, many of the searches to an index will be search for non-existing items. In an LSM-based index, such searches will be costly, because we have to search all the components C_i to be sure the item (in this case the OD) is not in the

index. However, in an OIDX, almost all searches are for existing items, so this is less of a problem than in the general case. However, if even a small percentage of lookups are for non-existing items, that can be enough to saturate tertiary storage bandwidth.

8.9.2 Container Migration

Often, it is possible to know in advance that data will not be needed on-line. This is for example the case in SSDBs, where only summary data is kept on-line. When this is the case, the base data can be stored in containers which in their entirety are migrated to tertiary storage, together with their OIDX subtrees.

It is also possible to move an arbitrary subtree of the OIDX, together with the related objects. This solution is difficult to use efficiently, because it is difficult to know which subtrees will not be frequently accessed after migration to tertiary storage.

8.9.3 Other Alternatives

If we have enough secondary storage to store all current versions of the objects and an OIDX indexing the current versions, we can use a separate OIDX on tertiary storage for historical versions (see Section 8.3.5). It is also possible to use the index on tertiary storage for all vacuumed objects, and keep all non-vacuumed objects on secondary storage.

If we expect tertiary store accesses to be very frequent, another OIDX can be considered, for example TSB-tree, which separate current from past versions in a possibly better way. However, this has its disadvantages, as discussed in Section 8.3.2.

8.10 Summary

OID indexing is one of the most important design issues in a temporal ODBMS. Every object update creates a new object version that must be indexed, making the OID indexing a potential bottleneck. We have in this chapter studied how to do object indexing in a temporal ODBMS, and proposed a new indexing structure suitable for this task, the *Vagabond Temporal OID Index* (VTOIDX). The use of the VTOIDX, together with an OD cache and the PCache which will be introduced in Chapter 9, should reduce the OIDX access cost to an acceptable level.

Chapter 9

The Persistent Cache

In a temporal ODBMS, the OIDX has to be updated every time an object is updated. This is a potential bottleneck, and in this chapter, we present the *persistent cache* (PCache), an approach that reduces the index update and lookup costs in temporal ODBMSs. In our paper in Appendix E we present a cost model for the PCache, and use this to show that the use of the PCache can reduce the average OIDX access cost to only a fraction of the cost when not using the PCache.

9.1 Introduction

The ODs in an OIDX have in general a low locality. The OD cache makes read accesses efficient, but in a database with many objects, most of the ODs that are updated during one checkpoint interval[1] will reside in different leaf nodes. This low locality means that *many leaf nodes have to be updated* during one checkpoint interval. When an index node is to be updated, an installation read of the node has to be done first. With a large index, the access to the nodes will be random disk accesses, and as a result, the installation read is very costly.

To improve performance, the *persistent cache* (PCache) can be used. The PCache contains a subset of the entries in the OIDX. The goal is to have *the most frequently used ODs in the PCache*. In contrast to the main-memory cache (the OD cache), the PCache is persistent, so that we do not have to write its entries back to the OIDX tree during each checkpoint interval. This is actually the main purpose of the PCache: to provide an intermediate storage area for persistent data, in this case, the ODs. The result should be a reduced update and lookup cost for ODs.

The size of the PCache is in general larger than the size of the main memory, but smaller than the size of the OIDX tree. The contents of the PCache are maintained according to an LRU like mechanism. The result should give higher locality on accesses to the PCache nodes, reducing the total number of installation reads. The average OIDX lookup cost will therefore also be less than without using a PCache.

To avoid confusion, we will hereafter denote the OID index tree itself as the *TIDX*, and use *OIDX* to denote the combined index system, i.e., the PCache and the TIDX. Thus, when we say an entry is in the OIDX, it can be in the PCache, in the TIDX, or in both. This is illustrated in Figure 9.1.

[1] A checkpoint interval is the time between two consecutive checkpoints.

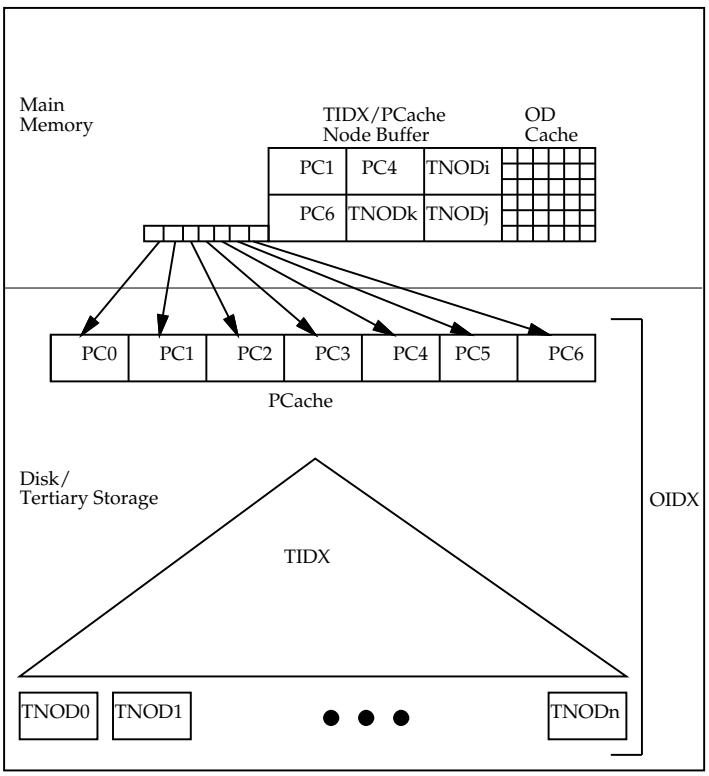


Figure 9.1: Overview of the TIDX, PCache, and index-related main-memory buffers. PCache nodes PC1, PC4 and PC6, and three TIDX nodes (denoted TNODn) are in the main-memory buffer.

9.2 PCache Organization

The index-related main-memory buffers, the PCache, and the TIDX, are illustrated in Figure 9.1. The number of nodes in the PCache should be small enough to make it possible to store pointers to all the PCache nodes in main memory. This means that even with a no-overwrite policy, the PCache could be implemented as a hash index, because no index tree over the buckets is necessary. However, to be able to do the copying of the ODs from the PCache to the TIDX efficiently, the nodes in the PCache should be accessed in the same order as the leaf nodes in the TIDX. Therefore, the nodes in the PCache are range partitioned, each node stores a certain interval of OIDs.

Range-partitioning is vulnerable to skew. To avoid this, the partitioning can be dynamically changed (for each node, we have the OID range boundary in main memory). The partitioning is done based on the update access rates on each node. A high update access rate to a node results in a smaller interval being allocated to that node. Because the PCache nodes are frequently accessed, the repartitioning does not represent any extra cost (the repartitioning can be done when neighbor nodes are resident in the buffer).

As described in Chapter 8), the TIDX itself is realized as a nested-tree index. The nodes in the PCache, however, are similar to nodes in a composite index (see Section 8.3.2). The drawbacks of using a composite index do not apply to the PCache, because the ODs in the PCache are the most frequently accessed ODs, mixing current and historical ODs is not a problem, and problems related to tertiary storage migration of the OIDX can be solved, an issue which will be discussed in Section 9.8.

9.3 LRU Management

The PCache nodes are operated similarly to an ordinary main-memory cache, and when an entry is to be stored in a node in the PCache, one of the existing entries has to be discarded. To provide access statistics, an access table is maintained for each node, with one bit for each entry in the node. The access bit is set each time the entry is accessed, and we use the clock algorithm as an LRU approximation. We have several alternatives for the storage of the access tables:

1. The access table of a node could be stored *in the node itself*. A problem with this alternative, is that when a node is to be discarded from the main-memory buffer, and one or more entries have been accessed, the node needs to be written back to disk, even if no entry has been changed. This is not desirable. A variant of this strategy is to not write the node back if only the access table has changed. However, if this is done, the access map will not necessarily reflect read accesses done to the node, and it will be unreliable as a way to approximate LRU.

2. Access tables are maintained in main memory, for each main-memory resident node. When a node is discarded, i.e., due to buffer replacement, the table is discarded as well. A problem with this alternative, is that until enough accesses have been done to the entries, the bit map is unreliable as a way to approximate LRU, and the "wrong" entries might get discarded.

3. Access tables are maintained in main memory for *all* PCache nodes. One problem with this approach, is that one table for each node in the PCache is needed, but because the size of each table is small (one bit for each entry in the node), this will not represent a problem as long as the PCache is not too large. If the system crashes, the contents of the access tables will be lost, and wrong caching decisions might be made when the system is restarted. This only affects *performance* at startup time, the ODs of committed operations are always safe on disk. To

reduce the amount of wrong caching decisions at startup time, we store the access table in the node as well when the node is written to disk. Note that this differs from alternative 1, where the node is *always* written back when it is discarded, even when there are no updates to the ODs stored in the node.

Based on the observations above, we believe that maintaining access tables in main memory for *all* PCache nodes is the best approach. The access tables are small, and keeping them in memory is a small price to pay for increased PCache hit rate. The access tables, PCache pointers, and OID ranges for all PCache nodes are kept in a *PCache Status Table* (PCST).

In addition to the access tables, each node also contains a table to keep track of the status of the entries with respect to the TIDX. One *dirty* bit is needed for each entry. The dirty bit is set each time an entry is modified or inserted into the node. Only the dirty entries need to be written back to the TIDX, entries not marked as modified can be safely discarded when needed.

9.4 Update Operations

When employing the PCache, ODs resulting from object *updates* are *always* inserted into the PCache, never directly to the TIDX. Inserting an OD into a node, implies discarding another OD from the node, based on the LRU strategy. It is preferable to discard a non-dirty entry, so that a synchronous writeback of the dirty entry is avoided. To have a high probability of non-dirty slots in the PCache node, dirty entries in the PCache are regularly copied over to the TIDX itself, asynchronously in the background. This is done efficiently by mostly sequential reading of the PCache nodes, and mostly sequential installation read and subsequent writing of the TIDX nodes.

9.5 Object Creations

Object creations are still applied directly to the TIDX. An OD resulting from an object creation is an efficient append operation into the TIDX. In the case where a new OD is to be part of the hot set, it will usually be retrieved from the TIDX node before it is discarded from the buffer. Such ODs will on access be inserted into the PCache.

9.6 Read Operations

Read operations belong to one of two classes, (navigational) single-object reads, and scan operations.

9.6.1 Single-Object Read

Read operations are done by first checking if the entry to be accessed is in the OD cache or the PCache. If not found there, the TIDX itself is searched. The search in the PCache might result in one disk access if the actual PCache node is not in the main-memory buffer. When using range partitioning, there is only one candidate node, so that at most one disk access will be needed. Accessing the TIDX can result in one or more disk accesses if the TIDX nodes are not in the buffer.

When found, either in the PCache or the TIDX, the OD is inserted into the OD cache. If the OD was not already in the PCache, we now have several options, for example:

1. Insert the OD into the PCache immediately. Note that at this point, we are guaranteed to have the candidate PCache node resident in buffer, because we have probed it during the search for the OD. If we manage to get a high hit rate on the PCache, the optimal OD cache size can be quite small in this case. Note that in addition to the entries inserted into the PCache, the OD cache also contains dirty entries from update operations that have not yet been installed into the PCache.

2. Insert the OD into the PCache only when it is to be discarded from the OD cache. In this way, we get good memory utilization, we do not have to use space for the OD both in the OD cache and in the PCache. However, in this case, we are not guaranteed to have the candidate PCache node resident in buffer, it might have been discarded since it was probed.

3. Never insert the OD into the PCache, only insert entries into the PCache when doing update operations. In this case, we rely on the OD cache to keep the most frequently accessed ODs, and use the PCache to be able to do efficient update of the TIDX. This strategy delays the update of the TIDX, and means that more entries can be collected before batch updating the TIDX.

Which option to choose, depends on the access pattern. The possible installation read of option 2 can make it costly, and because ODs retrieved from read operations are not inserted into the PCache in the case of option 3, we expect option 1 to be the most interesting. However, testing is necessary to verify this.

9.6.2 Scan Operations

Scan operations must be treated differently from single-object read operations, because one single scan operation can cause the current contents of the whole PCache to be discarded. The ODs retrieved during a scan operation will in general have less chance of being used again, it is not likely that the whole collection or container to be scanned, represents a hot set. Even if this is the case, it is possible that the number of ODs retrieved during the scan, is larger than the number of ODs that fits in the PCache. In this case, if we do a new scan over the collection/container, we will have a PCache hit probability of 0. This is similar to general buffer management in the case of scan operations.

As a result, scan operations should not update the PCache, but the PCache must be consulted during read, because recently updated ODs from the actual container/collection might reside in the PCache. However, this will not be very costly, because the contents of a physical container cached in the PCache will be clustered in the PCache's pages as well (because a physical container actually is a range of OIDs), so that the extra cost of reading the relevant PCache pages is only marginal.

9.7 PCache-to-TIDX Writeback

The update of the TIDX, i.e., writing dirty entries in the PCache into the TIDX, will be done in the background. This is done by reading PCache nodes, and installing the dirty entries in these nodes into the TIDX. This is done in segments, i.e., a number of nodes, and will be mostly sequential reading and writing. The PCache-to-TIDX writeback is a scan operation, and to avoid buffer pollution, nodes accessed during this operation should not affect the rest of the buffer contents, i.e., they should not cause other nodes to be removed from the buffer.

The rate of the writeback is a tuning question. By giving it higher priority, i.e., doing more frequent writebacks of PCache nodes, the probability of a PCache node being full of dirty entries is

less likely. This is important, because it reduces the probability of synchronous writebacks. On the other hand, higher priority to the writeback also means that more of the disk bandwidth will be used for this purpose, because each node contains a smaller number of dirty entries.

All ODs updated since the penultimate checkpoint, and still dirty in the OD cache, need to be installed into the PCache or TIDX during one checkpoint interval. This is not the case with the PCache-to-TIDX writeback. The interval between each time the contents of a particular PCache node are written back can be very long, but still short enough to avoid overflow of dirty ODs in the PCache.

9.8 PCache and TIDX on Tertiary Storage

ODs from TIDX nodes residing on tertiary storage may also be cached in the PCache. This can be very beneficial in the case when only a very few entries of in a large subtree of the TIDX are frequently accessed. In this case, the TIDX subtree can reside on tertiary storage, while the frequently accessed ODs reside in the PCache.

If objects to be updated only have the OD of the current version in the PCache and on tertiary storage, the nodes on tertiary storage have to be retrieved and rewritten (see Section 8.4.2) when the ODs are inserted into the HVOIDX.

9.9 Summary

We have in this chapter described the PCache, which reduces the OIDX update and lookup costs in temporal ODBMSs. This is achieved by clustering hot spot ODs together on PCache nodes, which increases the effective memory utilization, as well as reducing the number of pages that needs to be read and written.

The PCache has similarities to LHAM [143] (see Section 5.4.3), where a hierarchy of indexes is used. Important differences are that in LHAM, *all* entries in one level are regularly moved to the next level, there is no LRU management, and as such, it only helps to improve the write efficiency, not the read efficiency.

Chapter 10

Large Objects in Vagabond

Objects in Vagabond are very flexible. They have a dynamic size, and can grow and shrink from version to version without any performance penalties. When an object grows larger than a certain size, the *large-object threshold*, the object is converted to a *large object*. A large object is composed of subobjects and a subobject index, and the subobject index can contain additional information to be used in special objects. To give a high degree of flexibility, the large-object threshold can be set independently for each object class.

Different types of objects can have different access characteristics. Two examples are index objects (a general index stored as an object) and video objects. An index will often have a high rate of updates to small parts of the index object (to index entries), while a video object will be read only. Further, the index usually has random references to items, while a video will be read sequentially.

It is obvious that trying to unify the handling of all object types is likely to give low performance, a very complex system, or most probably both. In Vagabond, all non-trivial objects are called *special objects*. They are handled by *special object handlers* (SOH). In this chapter, we first describe how SOHs can be used to give Vagabond extensibility, flexibility, and improved throughput. Second, we describe how large objects can be used to implement indexes, collections, and multidimensional and spatial data structures.

10.1 Why Special Object Handlers?

In traditional DBMSs, many techniques are employed to increase the throughput of the system (number of transactions per second). However, even in the case when all data and indexes can be held in main memory, there are potential bottlenecks, the most important are the log-writing bottleneck, and blocking caused by concurrency control conflicts.

Log writing reduces the performance of a system for several reasons. One reason is the amount of data that has to be written, the second is the delay from a write request is issued until it is completed. To reduce the amount of data that has to be written to the persistent-log storage, it is possible to use compression (see Section 7.2) and techniques similar to delta objects (see Section 6.3.5). The log write delay can be reduced by using persistent storage with smaller access time. One such device is persistent main memory, e.g., main memory with battery backup. Another technique, which can be employed in a parallel system where the server nodes have independent failure modes, is logging to a neighbor server node, as is done in the ClustRa database management system [93]. The other common bottleneck is blocking caused by concurrency control conflicts, which in particular is a problem in high performance systems. The effect of this bottleneck is especially evident in the case of heavily

accessed indexes.

In Vagabond, index structures are stored physically in the same way as ordinary objects, but unlike ordinary objects, there are some issues that need careful attention:

- Update of only small parts of the objects, for example, the update of an index entry, should not result in a synchronous write of a subobject at commit time, this will hurt performance. Also, if a subobject is updated by several concurrent transactions, there should not be one write of the subobject for each of the transactions, rather, there should be something similar to logging of the updates for each of the transactions, and only one write of the complete subobject.

- Fine-grained concurrency control granularity is needed. If one transaction accesses a subobject, this should not block other transactions from accessing it, provided that they adhere to some concurrency control strategy. Index structures are frequently hot spots, and often non-2PL concurrency control techniques are used.

If treating special objects as ordinary objects, we could simply write delta objects in the case of index objects as well, in this way, ordinary objects and special objects are treated in a uniform, transparent way. In this case, the use of delta-objects works as a kind of logging. However, there are some problems with this approach:

- If we view delta-object writing as logging, this is similar to *value logging* (physical logging). The problem with value logging is that some operations generate more log information than strictly necessary, for example in the case of insert operations, where we might have to rewrite the whole index node. If *logical logging* was used, a simple "insert entry" log record would suffice.

- Non-2PL concurrency control is difficult to employ. In a uniform approach, we can not use knowledge of the contents and semantics of the special objects. For example, consider a B^+-tree implemented as an object. In a uniform approach, we would end up with locking large parts of the tree on every access. If we used the knowledge of the tree structure, we could use this to reduce lock contention.

- Different special object types need to be treated differently. For example, the `NavigDesc` field (see Section 6.2.1) is interpreted differently for different special object types, as they are used to do the traversal to find the relevant subobject. Thus, different index types need to be treated differently as well.

- Access patterns are different for different types of objects.

These problems necessitates a more flexible approach. We solve this by using *special object handlers* (SOH). When employing SOHs, there is one SOH for each special object type, and all accesses to special objects are performed through an SOH. If support for a new type of special object is desired, for example a new index type, all that is necessary is to write a new SOH. All SOHs must have some access interface methods in common, including methods for concurrency control and recovery.

The use of SOH gives a high degree of flexibility and extendibility. The idea of doing it this way is not new, variants of this approach is found in several *extendible database management systems*, including POSTGRES [195, 199, 200], BOSS [119], and DataBlades/Cartridges in commercial systems.

10.2 Special Object Handler Services

The responsibilities of an SOH include:

- Physically creating special objects.
- Retrieval and updates of special objects.
- Recovery.
- Concurrency control.

Note that in the case where high concurrency is not needed, an SOH can have a very low complexity, as it only acts as an interface to the large objects. In this case, its only function is to interpret the value of the `NavigDesc`, and to do the traversal of the subobject index in order to find the relevant subobject.

10.2.1 Create Special Objects

The first step of creating a special object is to create an object through the storage manager. This is similar to creating an ordinary object. Next, the relevant data structures in the special object are initialized.

10.2.2 Update Special Objects

When delta objects are created for ordinary objects, the delta objects are physical delta objects. An SOH is free to use its knowledge of the semantics of its objects to create logical delta objects. This is similar to logical logging in general. The SOH also has the responsibility of deciding when to install the delta objects into the objects.

10.2.3 Retrieve Special Objects

When accessing a special object, often only a part of this object is needed. When accessing an ordinary large object, it is addressed by physical location. This is not necessarily the case for a special objects. For example, in an index a key is used to find the relevant part of the index (the subobject) by traversing the index. This traversal is done by the SOH based on the value of the `NavigDescs`.

10.2.4 Recovery

The SOH also has the responsibility of processing the relevant logging information during recovery after a failure. Logging information is in this context the delta objects. This can be implemented with a very low complexity, because a no-overwrite strategy is used. In most cases, this is simply done by processing the delta objects created by the SOH.

10.2.5 Concurrency Control

The SOH has the responsibility of employing the relevant concurrency control techniques when accessing the special objects. This could be as simple as locking the whole object, or when high concurrency is needed, using more complex locking techniques (for example in the case of a tree based index).

10.3 Examples of Special Objects

In this section we describe how the most common special objects can be implemented.

10.3.1 Indexes

As described in Section 8.5.2, subobject-index nodes in the Vagabond large objects have a variable-sized `NavigDesc` field that can be used to implement indexes. Many of the frequently used database indexes, including B-trees, R-trees, and TSB-trees, are multi-way trees where the entries in the internal nodes are tuples of the form `(NavigDesc,pointer)`. In B-trees, `NavigDesc` is a key, in R-trees, `NavigDesc` is a n-dimensional rectangle, etc.

10.3.2 Collections

A collection is a collection of objects, with supporting methods for inserting, removing, and testing for the existence of a certain element. It also supports the use of an iterator to access the elements in a collection. The collections described in this chapter, are collections as described in the ODMG standard [43]. The ODMG collections are based on the base class **Collection**, and include sets, bags, arrays, and lists:

- Set: A set is an unordered collection of elements, *with no duplicates allowed.*. It extends collections with set testing operations and methods to create new sets based on the union, intersection and difference operations.
- Bag: A bag is an unordered collection of elements, where duplicates *are* allowed. It extends collections with methods to create new bags based on union, intersection and difference operations.
- Array: An array is a collection of a fixed number of elements that can be located by position, similar to an one dimensional array in C-like languages. An array can be resized, but this has to be done explicitly.
- VArray: Most systems also support variable-sized arrays. VArrays are dynamic, so that if an element is inserted at the upper end, the array will automatically be extended by one element, an explicit resize operation is not needed.
- List: A list is an ordered collection of elements. It extends collections with methods for replacing, removing and retrieving elements in the list, and has methods for concatenating and appending lists. It is important to note that elements in an ODMG list can be indexed by position, and that a list can be seen as an array where it is possible to insert elements not only at the end, but also between elements.

If a uniform representation is a goal, all collections can be represented by B-trees. In at least one commercial system, O_2, all collections except arrays are implemented in this way [166].[1]

For both sets and bags, a B-tree is very suitable. For sets, duplicate checking would be very costly if no efficient access method to the structure is provided. A B-tree will also be suitable in order to be able to search for elements. In the case where such operations are less used, the availability of other

[1] Although arrays in memory are represented as regular C++-arrays.

implementations of bags should be useful. The fact that B-trees can have low space utilization can affect performance of operations in these cases.

For arrays, we expect the operation of testing for the existence of a certain element will be less important, and we propose to implement the array in the same way as main-memory arrays are implemented. An array can change size, this does not generate additional problems, as large objects can change size dynamically. Therefore, VArrays and regular arrays will be implemented in the same way.

A list can be seen as an array where it is possible to insert elements not only at the end, but also between elements. Vagabond large objects can have inserts, and lists can be implemented similarly to arrays. However, after inserts have been done, we are not guaranteed 100% space utilization in the large objects, as is the case with arrays.

10.3.3 Multidimensional Arrays

Multidimensional arrays are not well supported in current systems. However, as is evident by the existence of systems for multidimensional arrays already on the market, support for multidimensional arrays inside the database system will be required in the future.

Dense multidimensional arrays can be supported directly by using regular large objects, with a suitable size of the subobjects. Storage of sparse arrays can benefit from using the `NavigDescs` in the subobject index to reduce the space needed for the arrays, only chunks with non-null data need to be stored.

10.3.4 Spatial Data Structures

A number of multidimensional-spatial indexing techniques can also easy be integrated into the system. We have already mentioned R-trees, another relevant class of indexes is quad trees [179].

10.3.5 Indexes, Collections, and the Temporal Aspect

Indexes and collections are represented as objects, and like all other objects, they can be versioned. It is possible to retrieve an index as it was at a certain point in time. Note that this is different from indexing temporal data. A general temporal index is used to make it possible to retrieve objects valid at a certain time.

To our knowledge, the use of versioned indexes has not been given attention previously. One reason for this, is probably that most research on temporal databases has been done in the context of the relational model. In this model, we do not have the concept of object identifiers, and an object can not exist outside a collection (set).

We believe versioned indexes can be useful, and they can be easily supported by Vagabond when indexes are represented as objects. However, the versioning cost will be horrible, and a more explicit approach, for example using a multidimensional index might be more suitable for this purpose.

10.4 Summary

We have in this chapter described in more detail the management of large objects in Vagabond. When large objects are used to implement index structures, it is very important to be able to do fine grained accesses to these objects. In this section, we described how this is done by the use of special object handlers. We also described how common index structures and collection types can be implemented in Vagabond.

Chapter 11

Temporal Object Declustering

Objects in Vagabond can be declustered over a set of servers in a server group, and server groups can be connected in a distributed system (see Section 6.4). Placement of objects on server groups is based on locality, as is common in traditional distributed ODBMSs. The objects to be stored in a server group are declustered on the servers in the group according to some declustering strategy.

The intention of this chapter is to provide an overview over related work and some possible declustering strategies. Temporal-object declustering is an area that in itself contains enough problems to warrant a separate thesis. With this in mind, we will only scratch the surface in this chapter, and reserve more in-depth research and analysis for further work. However, a simple qualitative analysis of the strategies is provided in Chapter 15.

We will first give an overview of related work on object declustering. We will then describe declustering of objects in a server group, and outline declustering in a distributed system, which we consider to be orthogonal issues.

11.1 Introduction

Figure 11.1 illustrates the evolution of objects with time. In a parallel system we want to decluster the objects over the servers. In order to avoid some kind of directory lookup when retrieving objects, the objects are declustered over the servers based on some mapping based on OID, timestamp, or both.

When retrieving an object version, we may want to retrieve the current version of an object, or to retrieve the version of an object that was valid at a particular time. If emphasis is on low-cost retrieval of the current object versions, a declustering based on hashing of the OID is very suitable. However, in a transaction-time temporal ODBMS, we often want to operate on a snapshot of the database, i.e., on a consistent version of the database as it was at time T_i. This is called a timeslice operation, and is illustrated on Figure 11.1, where all object versions that were valid at time $T = 6$ are hatched.

If communication cost is an expected bottleneck in a system, it is important to decluster the object versions in a way that minimizes communication as well as balancing the load. Two such strategies are illustrated on Figure 11.2.

To the left in Figure 11.2, a declustering based on hashing of the OID is illustrated. In this case, the hash value of the OID is used to determine on which node to store an object version. This gives efficient and predictable access to all object versions, but when operating on a snapshot this approach can be less efficient. As is illustrated, object versions valid at a certain time will in general be stored on different nodes.

To the right in Figure 11.2, a declustering based on timestamp is illustrated. This declustering

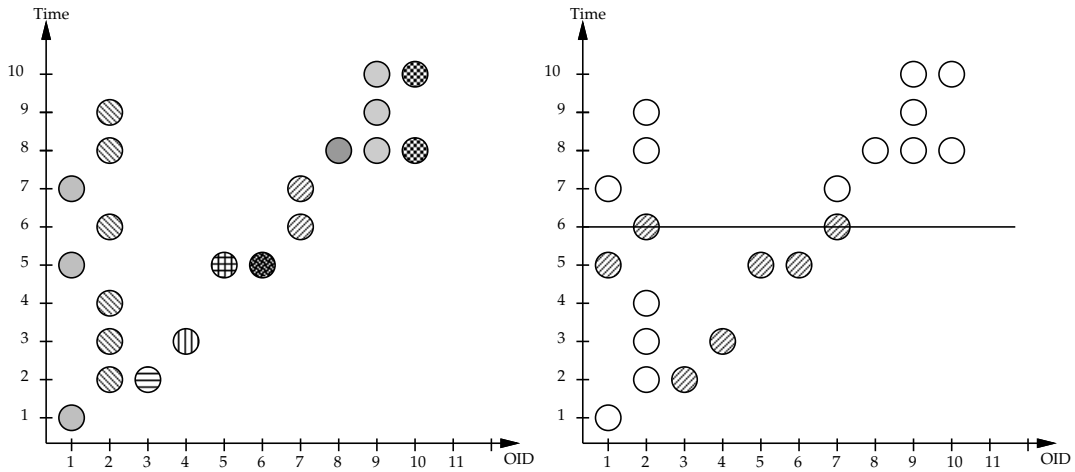


Figure 11.1: Object versions versus time. To the left, the circles denote object versions, and versions of the same object have the same hatching/coloring. To the right, we have illustrated a timeslice, by hatching all objects versions valid at time $T = 6$.

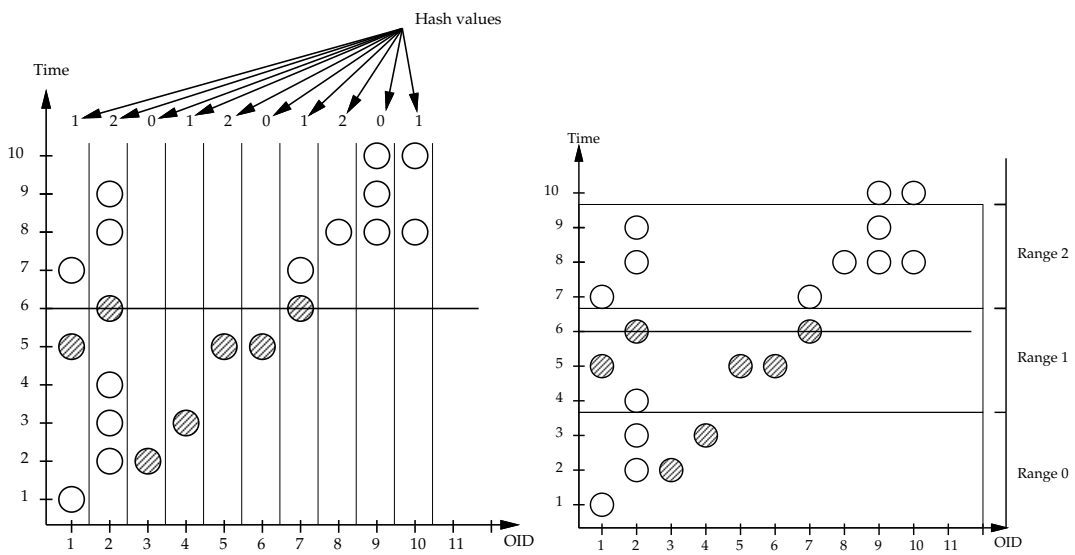


Figure 11.2: Two declustering strategies. To the left a declustering based on hashing of the OID, to the right a declustering based on range partitioning of the timestamp.

increases the probability that object versions valid in a certain time interval are stored on the same node. In the figure, we see that most of the objects valid at time $T = 6$ will be stored on the same node if this declustering strategy is used. If the (sub)transaction accessing these object versions is run on the same node, we reduce the communication costs. Such a declustering is also useful for other operations that can benefit from a declustering where object versions valid at the same time are stored on the same node, for example:

- Temporal selection: Many temporal queries involves a selection on time on a collection of objects (this includes objects valid at a certain time, or objects valid in a particular time interval). If objects valid at the same time are stored at the same node, the transfer volume can be significantly reduced. This is similar to parallel selection queries in non-temporal databases.
- Temporal grouping: This is also an example where the transfer volume can be significantly reduced if objects valid at the same time are stored on the same node. In an aggregate query with temporal grouping, most of the objects in a group are already on the same node.

11.2 Related Work

To reduce the query costs, optimal allocation and fragmentation is very important, but complex objects, object classes and inheritance, increase the size of the solution space for the data distribution problem in an ODBMS. In a temporal ODBMS, the aspect of time makes this problem even more difficult. Although these issues have been studied by a number of researchers, the corresponding update costs have not been studied, and experiments have been done on relatively small databases. This leaves a lot of questions unanswered. We will now give a brief overview of the most relevant work on object declustering in non-temporal ODBs and in parallel temporal database systems in general.

Non-Temporal ODBMSs. Declustering based on the hash value of the OID has been known to perform well in an ODBMS where set-based operations are common [174]. However, declustering based on hashing of the OID is not guaranteed to perform well for applications with an access pattern that is based more on pointer navigation.

To reduce the cost of pointer navigation, class-wise fragmentation can be used. Chen and Su [48] describe an heuristic partitioning approach based on class-wise fragmentation and allocating these classes, one or more, to each node. This is mainly interesting in databases with small class sizes and a large number of classes compared to the number of nodes. With a large number of objects in each class, this approach can easy give load balancing problems.

Another fragmentation approach has been proposed by Ghandeharizadeh et al. [76]. They use greedy algorithms to place all objects on nodes in an optimal way. However, these algorithms are very workload dependent, and for these algorithms to work, 1) access statistics are needed, and 2) a costly index lookup is needed to determine on which node a particular object is located if the fragmentation should be dynamic. We also question the scalability of the algorithms. The paper says little about the CPU cost, and in the experiments reported, a very small database with only 19531 objects was used.

There has also been done some work on object declustering in the context of distributed databases [9].

Parallel Temporal DBMSs. In a study of temporal query processing and optimization in multiprocessor database machines (in the context of a temporal relational database), Leung and Muntz [124] range-partitioned the tuples based on the timestamp. In a study of parallel query processing strategies

for temporal ODBMSs, Hyun and Su [94] used class-wise fragmentation, similar to the approach used by Chen and Su [48] described above.

11.3 Object Declustering in Server Groups

The object declustering problem has much in common with the traditional object clustering problem. In both cases, the existence of applications with very different access patterns makes it difficult to predict which objects will be accessed together, and should be stored close to each other. Clustering in temporal ODBMSs is as yet "uncharted territory", and we suspect that problem will prove to be even harder. This makes us believe that instead of using large resources to try to cluster objects together, it is better to simply distribute data as evenly as possible over the nodes, in a way that simplifies retrieval of current versions and keeps down the cost of timeslice operations. For other query types, we rely on data re-distribution during the query execution.

Declustering can be horizontal, vertical, or a combination. We will in this thesis concentrate on strategies for horizontal declustering, and in the rest of this section we will describe three different low-cost declustering strategies: 1) declustering based on the hash value of the OID of the objects, 2) range partitioning, based on the timestamp of the objects, and 3) a new hybrid algorithm, where current versions are declustered according to the hash value of the OID, and the historical versions are range partitioned based on timestamp.

11.3.1 OID-Based Declustering

In most ODBMSs, an object in a database is stored in a container/file, which can be logical or physical. A container identifier is often included in the OID, in addition to the unique number. To keep the discussion general, we consider an OID with the following attributes:

- *CONTID*: Container identifier, which identifies the container the object belongs to.
- *USN*: Unique serial number. Each object to be included in container *CONTID* gets an *USN* that is one larger than the previous *USN* allocated in the same container.

Simple OID-based declustering can be based one or both of these attributes of the OID. To be applicable, the strategy should decluster objects in a way that makes subsequent retrieval possible without the need for an additional (and costly) OID-to-node mapping index. We will now discuss the two strategies, and their characteristics.

Declustering on *OID*

When the *OID* declustering strategy is used, the node is determined by hashing the *USN*, or the combination of *USN* and *CONTID*, which has the same characteristics. In this way, objects will be evenly distributed over the nodes, and skew is unlikely to be a problem for queries that only access current versions of the objects. Figure 11.3 illustrates the *OID* declustering strategy in a system with $N_N = 4$ nodes. The node of an object is determined from the equation $OID\%N_N$, where % is the MOD operator.

This declustering strategy is especially applicable in the case of very large collections or sets, and when queries are mostly on the current versions of the objects. A typical example is aggregation, where a skew-free initial distribution is ideal for the first phase of a parallel aggregation involving

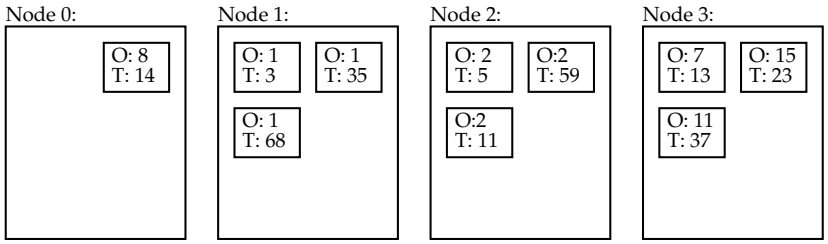


Figure 11.3: OID-based declustering. In the figure, an version is identified by the OID O:i and the timestamp T:T_i.

grouping, in which local aggregation is performed. In the second phase, the results from the local aggregation are redistributed, based on a hash partitioning value. However, there are two problems with this declustering strategy:

1. All versions of an object will be stored on the same node. If operations on historical versions of objects are frequent, for example historical aggregation, and the number of versions for each of the objects involved is skewed or only a small subset of the objects are accessed, we can get a load-balancing problem.

2. Versions of different objects that are valid at the same time will be on different servers, making timeslice operations expensive.

Declustering on *CONTID*

If only the *CONTID* is used as a parameter to the hash function, all objects that belong to a container will be stored on the same node. This also includes all the historical versions of the objects in that container. This can result in the same problems when doing queries on historical versions as when only the *USN* was used, and more importantly, the probability of a skew is very high when all the objects of a container are stored on the same node.

This declustering strategy is not as good for large sets as the *USN* only strategy. However, there are cases where declustering on the *CONTID* can still be beneficial:

- Operations on medium-sized collections, where member objects are processed together. One example is aggregate operations. With medium-size collections, the collections are not large enough to make parallel processing (on several nodes) beneficial.

- In cases where there are many references between the objects in a collection.

Because of the possible skew problem when using a *CONTID*-based declustering, we do not consider this strategy as appropriate, and do not discuss it further in this thesis.

11.3.2 Timestamp-Based Declustering

In a declustering strategy based on the timestamp[1] only, both hash partitioning and range partitioning can be used. If the timestamp is used as input to a hash partitioning function, we would effectively

[1]Note that in a transaction-time temporal database, we do not know the end timestamp when we create a new version. Therefore, only the start (commit) timestamp can be used, using the end timestamp is not an option as it would be in a valid-time temporal database.

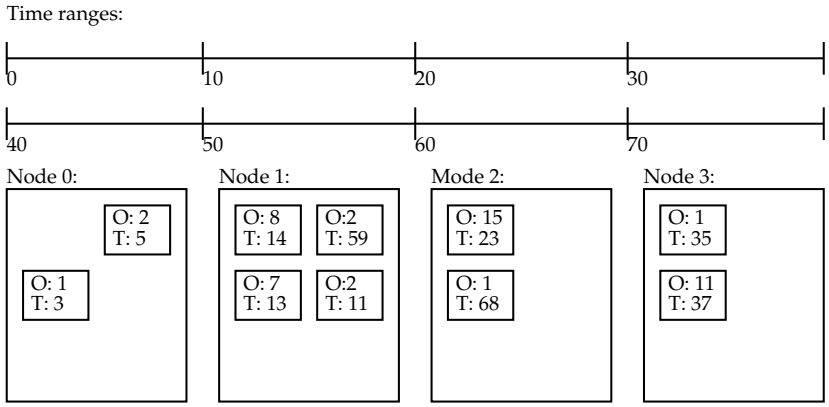


Figure 11.4: *TIME* declustering. On top of the figure, over the nodes, the time ranges covered by each node are illustrated. For example, node 0 covers the intervals from 0 to 9 and 40 to 49, node 1 covers the intervals from 10 to 19 and 50 to 59, etc.

distribute *all versions of all objects* over the nodes, with little skew. However, *we would not be able to know which node that contained a particular object*, as we would in most cases not know the timestamp of a particular version (note that this problem also applies to the current versions of the objects). To retrieve an object, broadcasting would be needed. This is very costly, and is not compensated for by any other benefits from declustering by hashing the timestamp.

Range partitioning based on the timestamp is a more interesting strategy. Although the problem with retrieving current versions of objects is still present, this problem is partially compensated by the increased probability of having objects with timestamp values close to each other on the same node [124], which is very beneficial when processing timeslice operations.

When range partitioning, hereafter called *TIME* declustering, is used, the number of time ranges should be much larger than the number of nodes, and these ranges should be distributed round robin across the nodes. The reason for this, is that time has an essentially unbounded value, as it is ever increasing, and that this strategy reduces the probability of skew. If many heavily accessed objects are created close in time, many short ranges instead of a few large ones reduces the probability of skew. However, at the same time, by increasing the number of ranges, and distributing them over the nodes, we gradually loose the possible benefits of timeslice operations, the probability of objects related in time are on the same node decreases. In this case, we have to make a tradeoff. This partitioning problem is similar to the partitioning problem in the *partitioned band join algorithm* proposed by DeWitt et al. [56]. DeWitt et al. used sampling methods to determine the range sizes, but in our context the problem is more difficult because the partitioning is not bound to the relatively short duration of a query. If sampling should be applicable in our context, it is likely that non-uniform range sizes should be used (at the expense of a lookup table to be able to determine which node covers a particular point in time).

Figure 11.4 illustrates the *TIME* declustering strategy. The node where an object version is stored is determined from the timestamp of the object. On top of the figure, we have illustrated the time ranges covered by each node. In this case, a node only covers two time ranges, but in general, the number of ranges will be higher. Note that even if an object covers (is valid) in more than one time range it is only stored once, on the node that covers its timestamp.

Object Access Operations

When declustering on timestamp, object access operations can be done as follows:

Create or update object: The object version is stored on the node determined by the commit timestamp. This has the advantage that everything created together is stored together.

Retrieve an object version valid at time T_i: When requesting the object version valid at time T_i, we first send a "get version valid at time T_i" message to the node N_i which includes the time range R_i =[T_j, T_k>, where $T_j \leq T_i < T_k$. An object version can cover more than one time range, and in that case it is possible that the object version was created in one of the previous time ranges. If the object version valid at time T_i was not created in the time range [T_j, T_i],[2] we have to search previous time ranges as follows:

1. The probe message is forwarded to the predecessor node. We "follow the timeline" when searching for the object version valid at time T_i, which means that we only consider objects created during R_{i-1} at this time. The message is forwarded until the actual object version is found, or all nodes have been probed. This can be illustrated with the following example, based on the declustering of objects on Figure 11.4:

   Example: Assume a search for the object with $OID = 1$ valid at time $T = 55$. We first probe node 1, but an appropriate object version is not found on that node, and we continue with node 0. The node contains an object version created at time $T = 3$, but we follow the timeline, and only consider object versions created during the time range [40, 50>. However, no object versions of object $OID = 1$ was created in that time range, so we continue with node 3 which covers the time range [30, 40>. Here we find an object version created at $T = 35$, and this is the desired result of the search.

2. One node will in general cover many time ranges. It is not necessary to forward the probe message several times through the "ring of nodes" to follow the timeline. To avoid this, we determine for each node we probe what is the most recent version of the searched object created before time T_i stored on that node. The timestamp of this version is included in the probe message. If the probe message already contains such a timestamp, the new timestamp replaces the existing timestamp if it is more recent than the previously determined most recent version. When the probe message has been through all nodes in the ring, and has reached the node which has node N_i as its predecessor, we are able to know which node stores the relevant object version. This can be illustrated by the following example, also based on Figure 11.4:

   Example: Assume a search for the object with $OID = 15$ valid at time $T = 75$. We start the search at node 3. However, no object version of the object with $OID = 15$ was created between [70, 75], so we have to probe the predecessor node, which is node 2. No object version of the object with $OID = 15$ was created in the time range [60, 70>, so we have to probe node 1. However, an older object version is stored on node 2, so that in the probe message to node 1 we also include the timestamp of the most recent object version stored on node 2, i.e., $T = 29$. No relevant object version is found on node 1, and the probe message is forwarded to node 0. This is the last node to probe. It contains no object version of the object with $OID = 15$, so

[2] Note that the node can also contain versions created in the time range <T_i, T_k>, but these versions are of no interest in this search.

that at this time, we know that the appropriate object version is the one with timestamp $T = 29$. A request message is sent to node 2 where this version is stored, and node 2 sends the actual object version to the requesting node.

3. It is also possible that the object was not yet created at time T_i. If this is the case, it can be determined from the probe message, where the timestamp of the most recent object version as described above will not have been set.

Retrieve the current version of an object: We do not know anything about the timestamp of the current version, so we have to probe all nodes to determine which node has the most recent version. This can be done in a number of ways:

1. First send a "get timestamp of most recent version of object i" message to all nodes, and then retrieve the object version from the node containing the most recent version.

2. Send a "get most recent version of object i" message to all nodes. All nodes return their most recent version of object i. In this way, we avoid some delay, but use more of the communication bandwidth.

3. Every time we create a new current version, we send a message about this to the node where the previous current version was stored (in general, when an object is updated, we know the timestamp, and implicitly the node, of the previous version). When we want to retrieve the current version of an object, we send a "get current version" message to all nodes. The node containing the current version knows this, so that only this node has to send an object back to the requesting node. We consider this to be the best strategy, as it keeps both delay and use of communication bandwidth to a minimum. This will compensate for the increased update cost.

A better alternative, especially if efficient broadcast is not supported, is to consider the retrieval of a current object version as the retrieval of an object version valid at time *now*. Assuming that most read current accesses are to the most heavily updated objects, this strategy will have a lower cost than involving all the nodes in the object retrieval as is the case when broadcasting is used. The drawback of such an approach is a possibly increased latency.

Possible Problems with *TIME* Declustering

The main problem with range partitioning is the same as with declustering on the hash value of the timestamp, we still do not know which node contains an object with a particular OID. Therefore, this strategy is only useful if timeslice operations are frequent compared to object navigation. Another problem in the context of a transaction-time database, is that all updates at a given time are done on one node, the node which contains the $[T_j, NOW>$ time interval.[3] In a system with high update rates, this node can become a bottleneck. The fact that many of the requests for current versions of the object will be satisfied by this node as well makes this problem very serious.

To reduce the cost of subsequent timeslice operations, it is possible to store an object version on all nodes whose timestamp range is (partially) overlapped with the time the actual object version was valid. This increases the storage cost, but by using a reasonable size of the timestamp range, the amount of replication does not have to be too high. However, every time we start on a new time range, i.e., writing to a new node, all objects that are still current have to be written to this node (but note

[3] NOW is the current time.

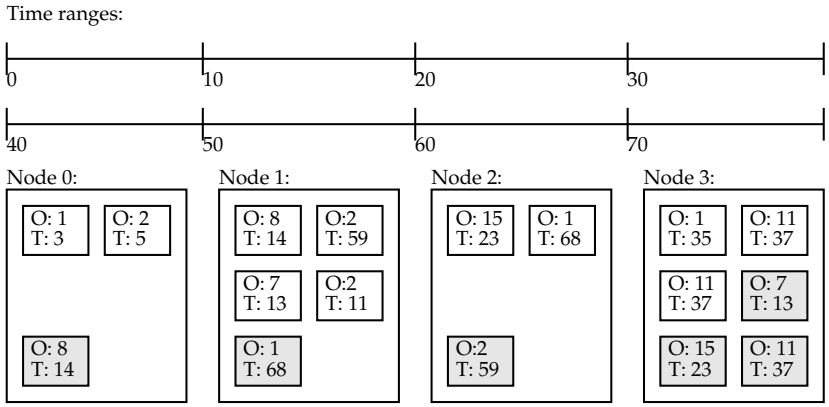


Figure 11.5: *OID-TIME* declustering. The replicated current version objects are illustrated by hatched boxes in the figure.

that an object version will only be written once to each node, so that in a system with N_N nodes, an object version that is very infrequently updated will be rewritten only during the first $(N_N - 1)$ time ranges after the version has first been written).

Replication is a very costly operation, and implies that timestamp-based declustering with replication is only beneficial if most objects are either updated very often or very seldom. In the first case, the object version is only written once, while in the other case, it will be written N_N times, on each node, but after it has been stored on all nodes, it will not incur any further replication cost. However, the storage costs in a database where data is not deleted is already very high, so we expect the additional storage costs from using replication to be unacceptable, and do not consider the use of replication in the analysis in this thesis.

Retrieving the history of a particular object can be more expensive with *TIME* declustering than with *OID* declustering, because the versions will be stored on different nodes. However, storing all versions on the same node can result in skew. Which strategy is best in this case, depends on what kind of operations will be subsequently applied to these versions.

11.3.3 Hybrid *OID-TIME* Declustering

We have now discussed declustering based on one of the attributes of the OID or the timestamp, and have seen that each of these strategies have both advantages and shortcomings. We will now outline a more suitable declustering strategy, the *OID-TIME* strategy, which aims at a skew free distribution, facilitates simple retrieval of the current version of objects, and keeps the costs of timeslice operations low.

The *OID-TIME* strategy is based on replication of the current version. When an object is created or modified, the new current version is stored both on the node determined by hashing the *OID*, and on the node determined by the range partitioned timestamp value. Retrieving the current version of an object is done by accessing the node determined by hashing the *OID*, and retrieval of a version valid at time T_i is done in the same way as with *TIME* declustering.

Figure 11.5 illustrates the *OID-TIME* declustering strategy. The node of an object version is determined from the timestamp of the object. In addition, the current version is stored on the node determined from the hash value of the OID. Note that once a new current version has been created,

the previous current version is not accessible on the node determined from the hash value of the OID, only on the node determined from the timestamp.

The *OID-TIME* declustering strategy carries an increased create and update cost, compared with the *OID* declustering strategy. To compensate for this cost, the amount of timeslice operations has to be sufficiently high. Similar to the timestamp-based declustering, all updates at a given time are done on one node. This means that this strategy is most suitable in systems with low or moderate update rates.

In a temporal database system, the database is partitioned, with current objects in the *current database*, and the previous versions in the other partition, in the *historical database*. When an object is updated, the previous version is first moved to the historical database, before the new version is stored in-place in the current database. When using *OID-TIME* declustering, the current versions declustered on OID will typically be stored in the current database, while the version declustered based on *TIME* will typical be stored in the historical database on that node. This implies than we do not need to move an object version when an object is updated, the previous current version is already stored in the historical database. When *OID* declustering is used, we need to move an object version every time an object is updated. This is an important point to note. In the cost analysis in this thesis we only consider inter-node communication, and not the cost of storing the objects in the nodes. Thus, in practice, the *OID-TIME* strategy will have a slightly better performance than the analysis shows.

11.3.4 Vertical Declustering

In some application areas, vertical declustering is desired. In the case of a server group, we expect this mainly to be the case for large objects. For some large objects, for example video objects, we want them to be striped over more than one server. This improves load balancing as well as throughput. This can be done transparently by striping the subobjects over the servers in a server group, or it can also be the responsibility of the user to partition the objects if he/she wants striping.

11.4 Object Declustering in Distributed Systems

We have now discussed the declustering of objects over the servers in a server group. Another issue is the declustering of objects in a distributed system. Although we expect that in most cases we will experience the same locality of historical object versions as the current versions, there will be applications were this is not the case. For example, it is quite possible that data is collected/generated at one site (server group), and the historical versions are used by other sites. However, in this case, we expect replication to be applicable. The use of timestamped objects makes this easier.

It is also possible to migrate the historical object versions. The OIDX supports forwarding, and this can be employed for objects that are very infrequently read at the site that writes them.

If signatures are maintained, it is also possible to benefit from caching of ODs in a distributed system. The signatures can be used in queries to reduce the number of objects that actually have to be retrieved.

11.5 Summary

The object declustering problem has much in common with the traditional object clustering problem. In both cases, the existence of applications with very different access patterns makes clustering difficult. This makes us believe that instead of using large resources to try to cluster objects together,

it is better to simply distribute data as evenly as possible on the nodes in the server group, in a way that simplifies retrieval of current versions and keeps down the cost of timeslice operations. For other query types, we rely on data re-distribution during the query execution. Chapter 15 presents a simple qualitative analysis of the declustering strategies discussed in this chapter. The analysis shows that similar to traditional database systems, an hash-based declustering (*OID*-based declustering) should be used in parallel temporal ODBMSs as well.

Chapter 12

Log-Only Database Operations

In this chapter we present the algorithms for the most important operations in Vagabond. We will start with an overview and an introductory example of log writing, and continue with more detailed descriptions of the different operations in the rest of the chapter. The description in this chapter is based on using magnetic disk as secondary storage. However, except for the device which stores the checkpoint blocks, other storage technologies can also be used.

12.1 Introduction

When a transaction is started, it is assigned a transaction identifier (TID). Unlike many other systems, it is not necessary to write any transaction start information to the log. In fact, if a transaction is aborted before it writes any objects to the log, there will be no trace left of the aborted transaction's existence at all.

An object that is written to the log, is always written together with its OD. Writing to the log is always done by writing large segments, which in general include objects from many transactions. The buffer system employs a steal strategy, so that modified objects can be written to the log before a transaction commits. This reduces commit time, as well as making it possible to handle large amounts of data in one transaction.

When objects are written to the log before the transaction has committed, we do not know the commit timestamp. Therefore, ODs written to the log contains the TID instead of the timestamp. To be able to know if an OD contains a TID or a timestamp, TIDs always have the most significant bit set. The ODs written together with the objects in the log are only intended to be used if crash recovery is needed. In order to be able to know the timestamp of a committed transaction when doing recovery, a `(TID,timestamp)` tuple for the committing transaction is written to the log as a part of the commit.

After a transaction commits, the objects that have been created or updated by the committing transaction become current versions. When another transaction later wants to read any of these objects, it have to first retrieve the OD of the object. This OD is either still in the OD cache, or has been installed into the OIDX. During normal operation, ODs are not discarded from main memory before they have been installed into the OIDX.[1] ODs from a committed transaction are lazily installed

[1] Remember that we denote the index tree itself as the TIDX, and use OIDX to mean the combined index system, i.e., the PCache and the TIDX. Thus, when we say an entry is in the OIDX, it can be in the PCache, in the TIDX, or in both (see Section 9.1).

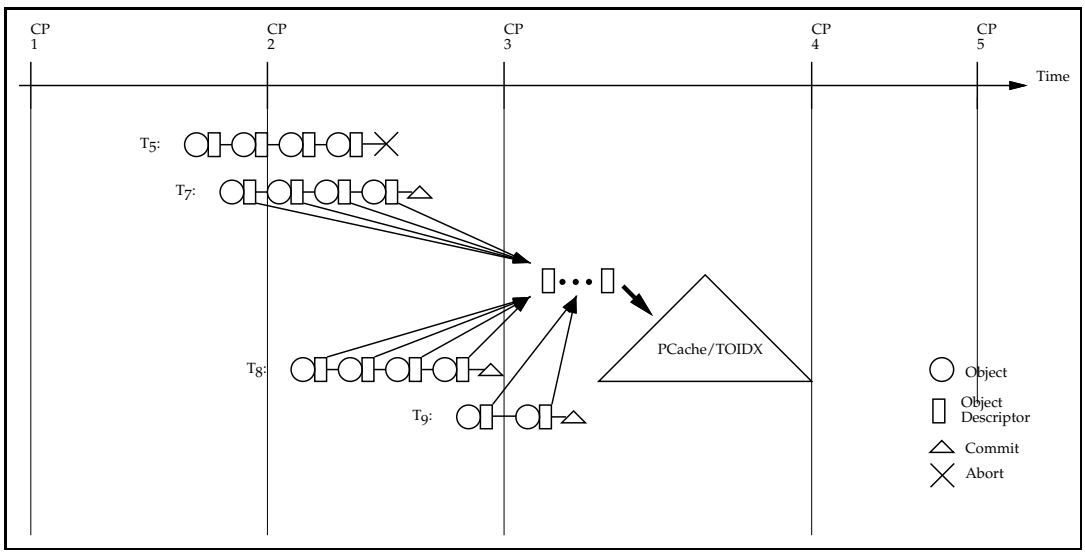


Figure 12.1: Example of log writing. In the figure, an object written to the log is illustrated with a ◯, an OD as a □, the commit operation by a △, and the abort operation with a ×.

into the OIDX after the commit has been finished. When the ODs are inserted into the OIDX, the timestamp is used in the OD, and not the TID.

Most transactions have a short duration, and will be short enough to make it possible to write all its created ODs, and do the completion of the commit operation, during one checkpoint interval (between two consecutive checkpoints). In this way, all its ODs are guaranteed to be installed into the OIDX during the next checkpoint interval. This means that during recovery, we know that we only have to process log back to the penultimate checkpoint.

If a transaction lasts longer than one checkpoint interval, we allow ODs generated by this transaction to be inserted into the PCache, even if the transaction has not yet committed. These ODs are stored as *uncommitted ODs* in the PCache nodes, and can not be used by any other transaction. In this way, when the transaction commits, all its ODs are guaranteed to be installed into the OIDX during next checkpoint interval. Some of its ODs in the PCache will at this point still be marked uncommitted, but during crash recovery we know which committed transactions have dirty entries in the PCache, so that these can be handled properly. Entries from aborted transactions will be lazily removed from the PCache nodes as they are retrieved from disk. Objects in the log from aborted transactions will simply be discarded the next time the segments are cleaned.

Crash recovery is simple in a log-only DBMS. If crash recovery is needed, the last part of the log is scanned. All ODs generated from a transaction that commits during one checkpoint interval, should be installed into the OIDX when the next checkpoint interval ends, so that an upper bound exists for the amount of log that has to be processed in the case of crash recovery. ODs from committed transactions that are not yet installed into the OIDX are collected, and can later be installed into the OIDX. ODs from aborted transactions and transactions that were ongoing at crash time, are ignored.

Example

We will now illustrate log writing with the use of Figure 12.1, which shows a number of transactions. On the top, we have the time-line, running from left to right, with checkpoints marked. The inserts of

ODs into the OIDX is illustrated with arrows in the figure. Please note the following:

- The length of a checkpoint interval depends on the workload.
- The commit operation in the physical log is composed of suboperations, i.e., prepare, commit, and commit completed. This will be explained in more detail in Section 12.3.
- Even though the transactions are illustrated with separate lines, objects and ODs from different transactions can be stored in the same segments.

Starting with transaction T_8, this is an ordinary short transaction. The transaction commits during the second checkpoint interval, and the ODs generated from this transaction are guaranteed to be installed into the OIDX when checkpoint 4 ends. In the case of a crash after checkpoint 4, no ODs from transaction T_8 will exist in the part of the log that have to be processed during recovery.

Transaction T_7 spans more than one checkpoint interval, and is therefore treated as a long transaction. All the ODs written by transaction T_7 during the first checkpoint interval, before checkpoint 2, must be installed into the PCache before the end of checkpoint 3. This will be done in the background during the second checkpoint interval. After the transaction have committed, its ODs can be inserted into the TIDX as well. To emphasize: Before transaction T_7 commits, its ODs can only be inserted into the PCache, but after the transaction has committed, its ODs can be inserted into the TIDX as well as the PCache.

Similar to the case of transaction T_7, some of the ODs from transaction T_5 written during the first checkpoint interval might have been inserted into the PCache before it aborts. If this was the case, they will be removed from the PCache in a lazy way, as time goes by. ODs (and objects) written to the log will be removed later, during the segment cleaning process.

Transaction T_9 commits during the third checkpoint interval, and its ODs might be inserted into the OIDX during the same interval, but this is not guaranteed to be finished until checkpoint 5 finishes, i.e., the second checkpoint after its commit.

We have now given a short introduction to the log generation, and we will now continue with a more detailed description of the operations.

12.2 Object Operations

A new OD is created every time an object is created, updated or deleted. In the case of an object create or update, the OD is written together with the object to the log, in the case of an object delete, it will be written to the log at a convenient time. In all cases, they will be written to the log before the transaction can finish the commit operation. The ODs will be inserted into the OIDX if the transaction commits. An OD will *never* be inserted into the TIDX itself before the actual transaction commits, but in the case of large transactions, some of the ODs can be inserted into the PCache (this will be described in more detail later in this section). Modified OIDX nodes will in general be written at a later time, so that the response time for a transaction commit can be short.

The following description of the operations is mainly independent of which concurrency control strategy is used. This means that we assume that in addition to performing the actions described below, concurrency control aspects are maintained. For example, if two-phase locking is used, we expect that the necessary lock(s) have been acquired before the actual operation is carried out.

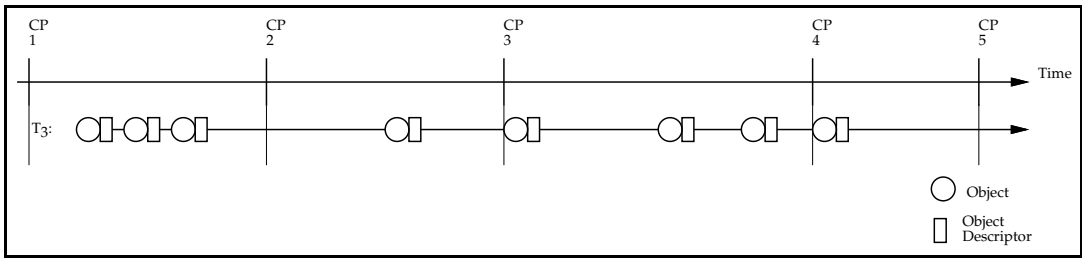


Figure 12.2: Long transaction.

12.2.1 OD Management

Most transactions are short and update only a few objects. The ODs they have generated are usually not discarded from main memory until they have been installed into the OIDX. The ODs written together with the objects in the log will only be used if crash recovery is needed. All ODs resulting from a transaction committed during one checkpoint interval, should be installed into the OIDX before the next checkpoint interval ends.

In the case of long[2] transactions, the number of ODs can be very high. If we require all ODs to be resident in the OD cache until they are inserted into the OIDX, the maximal size of a transaction would in this case be restricted by the size of the OD Cache. Another problem is transactions that last longer than a certain number of checkpoint intervals. *All* of the log created from the time when this transaction started to write to the log has to be processed during crash recovery. This can take a lot of time and it makes cleaning more complex, which in many situations is not acceptable.

These problems are illustrated in Figure 12.2, which shows an example of a long transaction. Note that nothing is written to the log from a transaction before the transaction commits or one of its objects (and OD) is evicted from the buffer. A transaction that only modifies a few objects, and accesses these frequently, does not necessarily write anything to the log even if it has a long duration. Thus, the transaction T_3 might have started long before checkpoint interval 1, because we only consider the time from the point the first object and OD is written to the log. If this transaction commits during checkpoint interval 5, and the system crashes shortly after that, all the log back to checkpoint 1 has to be processed during recovery. This is necessary because the log might contain ODs from this committed transaction.

Some possible solutions to the problems with long transactions are:

1. Use a larger OD cache.
2. Use one or more segments as "sidefiles" to store the ODs from long transactions.
3. Allow ODs from uncommitted transactions to be inserted into the PCache.

We will now discuss these alternatives, and why we consider alternative 3 as the best solution.

Large OD Cache

Solving the long transaction problem by increasing the size of the OD cache has several drawbacks:

[2]When we in this chapter study long transactions, we mean both traditional long-living transactions, as well as "large transactions" (transactions that generate large amounts of data).

- Increasing the OD cache means less memory available for buffering OIDX nodes and objects. This increases the buffer-miss rates, reducing the performance of the system.
- This approach is not truly scalable. Even in the extreme case of allocating all of the available main memory to the OD cache, there will be transactions that create more ODs than will fit in the OD cache.
- The problem with long-lived transactions is not solved, because there would still be a large amount of log to be processed during recovery. Most of the log would be from transactions that have already been committed and had their ODs inserted into the OIDX, which means that most of the log processing is duplicating earlier work. Most of the ODs in the log have already been inserted into the OIDX.

For these reasons, we do not consider this as an reasonable alternative.

Sidefile Segments

One or more segments could be used as "sidefiles", which are segments used only for storing the ODs generated by one transaction. The sidefile segments can later be processed efficiently if the transaction commits, by reading the sidefile segments and inserting the ODs into the OIDX. A transaction would start to write to a sidefile segment when the number of created ODs from the transaction becomes larger than a certain threshold, or when the transaction lasts longer than a certain number of checkpoint intervals.

By using sidefile segments, we avoid locking a large number of ODs in the OD cache, and reduce the amount of log that needs to be processed during recovery. However, as was the case with the previous approach, the sidefile segment approach has also got its problems.

One problem with sidefiles, is the situation where we have a long-lived transaction, that updates the same object more than once. If the time between each update of the object is long enough, the object will be written back to disk due to buffer replacement. In this case, the next time the object is modified, we do not know that it has already been modified by the same transaction. As a result, we will have several ODs representing updates on this object in the sidefile. Only the last OD should be inserted into the OIDX (one transaction creates only one new object version), so this is wasted disk space and disk bandwidth. However, although this problem is a nuisance, it is not fatal. By careful processing of the sidefile segments and inserts into the OIDX, duplicate ODs can be detected.

Another problem, is what happens after a transaction T_1 has committed. If this transaction generated sidefile segments, the ODs in the sidefiles are not yet in the OIDX, and they are not guaranteed to be in the OD cache in main memory. This means that if another transaction T_2 tries to access objects written by T_1, it is possible that the transaction can read a non-current OD, because it does not know that there is a more recent OD stored in the sidefile. This is certainly not acceptable. Again, this is a nuisance, but not fatal. The problem can be solved by not releasing write locks until after the sidefile segments have been processed. The unfortunate result of this is of course that objects can be locked for quite a long time. However, in most situations, this should not be a real issue. Long transactions will keep write locks for a long time anyway, which means that if this is regarded as a problem, the problem is likely to be present even without the locks on objects in sidefiles.

The really serious problem with sidefiles, is what happens when a transaction wants to read an object that has been previously modified by the same transaction. If the object and its OD has been replaced in the main-memory buffers, the transaction would need to search all of its sidefile segments to find out where the object is. This would be necessary to do for every read operation where it has

already got a write lock on an object whose current version is not main-memory resident, because it could not be sure if it had modified a certain object or not. Although this problem could be solved by storing the sidefile ODs in index trees, this would affect performance too much. Instead of simply writing the ODs sequentially into the sidefiles as described previously, a costly insert into a sidefile tree would be necessary. As a result, we consider using sidefile segments to be too complex and costly.

ODs from Uncommitted Transactions in the PCache

The third approach is to allow ODs from uncommitted transactions to be inserted into the PCache. This is not an ideal solution. However, it seems to be the most efficient and less complicated solution to our problems. PCache nodes are frequently read and written, so that the extra cost is only marginal. Because most transactions commit, the PCache space wasted by this approach is only marginal.

We do not allow ODs from uncommitted transactions to migrate further from the PCache to the TIDX. The reason for this, is that this would complicate commit processing, recovery and it would also be costly. Also, it should not be necessary. In the case of very long transactions, the size of the PCache can be adaptively resized, so that its size does not limit the transaction size.

It can look as if we loose the append-only advantage of inserting ODs from created objects if we first store them in the PCache. Luckily, this is not a bad as it seems at a first sight. When the ODs are created, they will be written sequentially into one or more PCache nodes. Later, when the transaction has committed, and the ODs are moved to the TIDX, they will be part of a sequence, so that the inserts of the ODs into the TIDX can be done very efficiently, in an append like way.

12.2.2 Management of ODs from Uncommitted Transactions

Based on the discussion above, it is clear that the best solution to the problems regarding long transactions, is to allow ODs from uncommitted transactions to be inserted into the PCache. In this case, we need to know which ODs in the PCache nodes are ODs from committed transactions, and which ODs are from uncommitted transactions. This is necessary in order to avoid other transactions accessing ODs from the uncommitted transactions, and because ODs from uncommitted transactions contain TIDs instead of timestamps. We can solve this in several ways, for example:

1. Use one bit in the OD as a flag to tell whether it belongs to an committed transactions or not.

2. Use a separate bitmap in each PCache node, to know which slots in the node contain ODs from committed transactions, and which slots contain ODs from uncommitted transactions.

3. The ODs in a PCache node are stored in a binary tree. In order to know which ODs are from committed transactions, and which ODs are from transactions that were uncommitted when the ODs were inserted into the PCache node, two trees can be used. One tree, the *committed tree*, is used for the ODs of committed transactions, and another tree, the *uncommitted tree*, is used for ODs from uncommitted transactions.

Using a separate tree in the PCache node is the best solution. Only one extra pointer in each PCache node is needed, so this solution has minimal space overhead. Management is also cheaper than for the two other approaches, because it is easy to find the ODs from uncommitted transactions when necessary.

When a transaction commits, the ODs it generated, and which have been inserted into PCache nodes, should be moved from the uncommitted trees to the committed trees. This is done lazily. Every

time a PCache node is retrieved, all ODs from committed transactions that are still in the uncommitted tree are moved to the committed tree. When an ODs is moved, the TID in the OD is replaced with the commit timestamp of the transaction that generated the OD.

We have to keep the TID of a committed transaction until all ODs stored in the PCache before it committed, have been moved to committed trees. For each committed transaction, we maintain a counter of how many ODs it still has in uncommitted trees in the PCache, and every time we move an OD from an uncommitted tree to a committed tree, we decrement this counter. When the counter reaches zero, we can discard information on this transaction.

The `(TID,timestamp,counter)` tuples are stored in a TID/timestamp/counter table (TTCT). The TTCT is written to the log during each checkpoint interval, in order to make it possible to reconstruct the table during recovery.[3] Most transactions will be small, and ODs from these transactions will not be written to the PCache. Therefore, the size of this table will be relatively small.

12.2.3 Creating and Updating Objects

When we create an object, it is allocated a unique OID, and an OD is created. A new OD is also created every time we update an object. The OD of the new version will eventually make its way into the OIDX if the transaction commits, as described in the previous section. There is little difference between temporal and non-temporal objects in the case of create and update operations, the difference is mostly whether the old OD is kept in the OIDX (see Section 8.4.2).

To ensure durability, an object *has* to be written to disk before its transaction commits. This is similar to a traditional system, where we need to have the necessary information written to the log before the transaction commits. The difference, however, is that in Vagabond, the log is the final place for the objects.

The buffer system employs a steal strategy, which means that a modified object can be written to disk before the transaction commits. The object and its OD is written together with other objects and their ODs, possibly from other transactions, into a segment. We do not know the timestamp of a transaction before it commits, so in the log we always use the TID instead of the timestamp in the OD. This is also the case for ODs of uncommitted transactions when they are inserted into the PCache.

When large objects are updated, only the modified subobjects and the affected subobject-index nodes are written to the log. The use of the Vagabond large-object index, presented in section 8.5.2 ensures that only the affected parts of the subobject index need to be written. Large objects are possibly spread over several segments, and therefore the writing of large objects has to be done carefully. This is done by first writing the updated subobjects, and then the modified parts of the subobject index. Note that the timestamps in SODs (see Section 8.5.2) are the time of segment write, not the commit time. The reason for this, is that in this way we do not have to update the subobject after the transaction has committed. The timestamps in the SODs are only used during cleaning, and for that purpose, the exact commit timestamp is not needed. This will be further described in Section 12.7.1.

When we update an existing object in Vagabond, we can either do the update on the current version in the buffer, or on a copy of the current version, and not on the current version itself. Making a copy increases the update costs, but has two important advantages that makes it beneficial:

1. If the update transaction aborts, the object is still in memory, so that we avoid reading it from disk if it is needed again. This is important, especially for hot-spot objects. Although aborts are

[3] Note that because we have to do the TID-to-timestamp mapping when we move ODs from an uncommitted tree to a committed tree, a "presumed commit" approach, where we only store the TIDs of aborted transactions, is not practical. We would still need to store the TID/timestamp mapping for each committed transaction.

rare, the cost of reading a single object back to main memory should justify this strategy.

2. If we did not make a copy, we would not have the previous version in memory, making a subsequent temporal operation more costly.

How beneficial this strategy is, depends on the abort rate, subsequent use of the object and the previous versions, and if delta objects will be created or not (to make a delta object, both versions are needed).

We will now describe in more detail the use of delta objects and compressed objects. The OD of an object version will contain the information about whether it is a delta object or is compressed (see Section 8.1.2).

Delta Objects

A delta object is the difference between the new and the previous committed version of an object (see Section 6.3.5). By writing delta objects instead of the whole objects, the amount of data that has to be written to the log is reduced. Delta objects in Vagabond are in general physical delta objects, i.e., the difference between the physical versions, but the special object handlers can create logical delta objects as well (see Section 10.1).

When a delta object is written, the *delta object* bit is set in the OD if the object is a small object. When writing a delta subobject is written, the *delta subobject* is set in the SOD.

The fact that only a delta object is written to disk does not affect the efficiency of future accesses to the current version of an object while it is still in the buffer. However, if the object is removed from the buffer because of buffer replacement or a system crash, future accesses have to read the last complete version that was written to disk, as well as the delta object(s) written after that, in order to reconstruct a particular object version. Reading a chain of delta objects is costly. For current object versions, it can be avoided by always writing the complete object to disk before removing it from the buffer, and when doing a system shutdown. This means that it is only after a crash that it will be necessary to retrieve the current version of an object through a chain of delta objects. When retrieving historical object versions, reading a chain of delta objects can be necessary if the actual version was only written as a delta object.

It is not always beneficial to write delta objects. This should only be done if the following criteria are satisfied:

- The size of the delta object should be much smaller than the size of the complete object. This is the most important criteria. If this is not satisfied, writing a delta object is of no use.

- There should be some probability of a new update to the object before it is discarded from the buffer, but it is difficult to know if this criteria is satisfied.

- Accesses to historical versions of this object should be infrequent. If this is not the case, retrieval of historical object versions will be too costly.

- In the case of non-temporal objects we do not normally keep previous versions, and only one OD is stored in the OIDX. However, if writing delta objects, one OD for the last whole version and one OD for each intermediate delta object have to be stored in the OIDX. Even though a few of the previous ODs can be cached in the CVOIDX node, too many delta versions makes HVOIDX searches frequent, even in the case of non-temporal objects. This should be avoided.

Compressed Objects

If we want to write a compressed object, this is done by first compressing the object, writing the result to a new location in the buffer, and then writing the object to the log. Together with the compressed object, we also write the uncompressed size of the object. The size stored in the OD is the compressed size of the object, i.e., how much space the compressed version occupies on disk.

After the compressed object has been written to disk, the compressed version of the object can be removed from the buffer. The uncompressed objects are kept, except when the reason for writing the object to disk was buffer replacement.

When compression is used on a large object, compression is done independently on the subobjects. If this was not the case, retrieving and updating single subobjects would be difficult and inefficient. Whether a subobject is compressed or not, is stored in the SOD (see Section 8.5.2).

12.2.4 Deleting Objects

Temporal objects are not physically deleted. In this respect, they are mostly treated as non-deleted objects. For example, during cleaning, a deleted object will be moved to the new segment, similar to any non-deleted object. A non-temporal object, on the other hand, can not be accessed after it has been deleted. It will be physically removed from the segment it resides in the next time the segment is cleaned. Whether an object is temporal or non-temporal, is stored in the object's OD.

Deleting Temporal Objects

Deleting an object which is defined as temporal, is done by writing a tombstone OD, which is an OD where the physical location is NULL, and the timestamp is the delete time.

Note that whether the object is a large object or not, does not matter for the delete operation when the object is temporal.

Deleting Non-Temporal Objects

If we do not want to keep the deleted version, i.e., it is not a temporal object, its OD is written to the log with both physical location and timestamp set to NULL. Unlike the tombstone OD, this OD is written to the log as logging information to be used in the case of recovery. When the OIDX is updated later, the OD for this object will be removed. When the object is deleted, the live-byte counter (in the SST) for the segment where the object resides, is decremented accordingly.

If the object is a delta object, the live-byte counter for the segment where the last complete object was written is decremented, and if there where intermediate delta objects, the live-byte counter for the segments where these delta objects reside is decremented as well. The object (and delta objects) will be removed next time the segment(s) are cleaned.

If the object is a large object, the subobject index has to be traversed in order to decrease the live-byte count for the segments where the subobject-index nodes and the subobjects are stored. The subobject-index nodes and the subobjects will be deleted next time the respective segments are cleaned.

Deleting New Objects

If an object is deleted by the same transaction that created it, the effect on the database should be the same as if the object had never been created. This is assured by using the following algorithm:

1. If the object has not yet been written to the log, the only action needed is to remove the OD from the OD cache and delete the object from the main-memory buffer.

2. If the object has been written to the log, but its OD is still in the OD cache and is dirty with respect to the OIDX, the only action needed is to write a tombstone OD to the log. If a crash occurs, the recovery algorithm will know that an object deleted by the same transaction that created it, should be discarded.

3. If the object has been written to the log, and the OD in the OD cache is clean or the OD is not resident in the OD cache, that means that the OD has been inserted into a PCache node. In this case, the OD has to be removed from the PCache node. In order to avoid a synchronous operation, a tombstone OD is created and inserted into the OD cache. The OD in the PCache node is removed the next time the PCache node is brought into main memory. The tombstone OD that is inserted into the OD cache has to be written to the log before or during commit, if the PCache node has not been updated before that time.

Reincarnation

Reincarnation is a concept used in only a few temporal-object models, and assumes support for non-contiguous lifespans. These models can also be supported in Vagabond. Reincarnation can be done by writing the reincarnated object together with an OD where the timestamp is more recent than the timestamp of the tombstone version.

Note that even if the reincarnated object is the same as the last non-deleted version, it is not enough simply to make a new OD where physical location points to the last valid version of the object. If we did it this way, we would have two ODs (possibly more, if an object was deleted and reincarnated several times) representing the same physical object. This would complicate cleaning and object moving considerably, because we would have to access and update more of the OIDX.

12.2.5 Reading Objects

We will now describe how to retrieve current as well as historical object versions, and how to treat delta objects and compressed objects.

Reading Objects Stored as Complete Versions

The physical location of an object version is stored in its OD (see Section 8.1.2), and in order to read an object that is not resident in memory, we have to first retrieve its OD. If the object is a large object, the location in the OD is the location of the root of the subobject index of the actual object version, and this subobject has to be traversed in order to retrieve the requested subobject(s).

Current Versions. When we want to read the current version of an object, we first check if the OD is resident in the OD buffer. If not, we have to do a lookup in the OIDX. An OIDX lookup is done by first searching the PCache, and if the OD is not found in the PCache, the TIDX is searched (see Section 8.4.2). When doing a lookup in a PCache node, it is only necessary to search the uncommitted tree in the PCache node if it is possible that the object has been previously modified by the same transaction that is now requesting the object. If a locking protocol is used, this can only be the case if the transaction already owns a write lock on this object (or in the case of hierarchical locking, a lock for a larger granularity, for example a container). If the OD is found, the object is read from the

physical location found in the OD. If the OD is not found, this means that the object is deleted, or has never existed. This error is typically caused by a bug in the application program, for example a reference to a deleted object.

Historical Versions. If we know the timestamp of the historical version we want to retrieve, the lookup for the OD and retrieval of the object can be done in the same way as when reading the current version of an object. However, quite often the query is for an object version valid at a certain time t_j. In this case, we have to retrieve the OD with *the largest timestamp less than or equal to* t_j. It is this operation that makes an end timestamp in the OD beneficial when the OD is outside the TIDX (see Section 8.1.2). If we did not have the end timestamp in the OD, it would not be sufficient to access the OD cache or the PCache to find the OD. Even if we found an OD in the OD cache or PCache with a timestamp t_i that is close to t_j, there could have been updates between t_i and t_j. This would be impossible to know from the ODs alone, and we would have to do a lookup in the TIDX for every such retrieval.

Delta Objects and Compressed Objects

As described above, we have to retrieve the OD before we can retrieve an object, in order to get the physical location where the object is stored. The OD also contains information on whether the actual object version is a delta object or is compressed, except in the case of a large object, where this information is stored in the SOD.

Delta Objects. If an object version V_j is a delta object, we have to retrieve the most recent complete version V_c that is older than V_j, in addition to the delta objects D_i written between V_c and V_j. This is done by first retrieving the ODs of the delta objects D_i and the complete versions V_c by searching the HVOIDX backwards, from version V_j. The number of delta objects between two complete versions should be relatively small, and the ODs will be clustered in the HVOIDX, so this lookup will not be very costly. After the ODs have been retrieved, the actual object version V_c and the delta objects are retrieved, so that object version V_j can be reconstructed.

Note that when delta objects have been written for a non-temporal object, there will be one OD for each delta version and one OD for the last complete object written. Except for the most recent, the ODs of the object will be stored in the HVOIDX, similar to the case of temporal objects (see Section 8.4.2).

Compressed Objects. If the requested object version was compressed before it was stored, the compressed version is read into the object buffer, and then decompressed into a new location in the buffer. After decompression, the compressed version is removed from the buffer.

12.2.6 Subsegments

If the system load is low, or transactions are mostly read-only, only small amounts of new data will be created. In that case, update transactions in the commit phase will experience long delays if we have to fill up the segments before we write. This is not acceptable and a solution to this problem is to write subsegments, as outlined in Section 5.1.2.

12.3 Transaction Management

In order to be able to do recovery after a failure, it is necessary to ensure that enough information has been written to the log before a transaction commits. We have previously in this chapter described how objects can be written to disk before a transaction commits, in order to avoid writing all the objects modified by the transaction in one burst during commit, and how we write the ODs to the log in order to avoid synchronous updates of OIDX nodes at commit time. We will in this section describe in more detail transaction management in Vagabond.

12.3.1 Commit

The transaction commit operation can in principle be implemented by writing objects from the transaction that is still dirty in the object buffer to the log, followed by a transaction finished mark which includes a `(TID,timestamp)` tuple. After the objects and the transaction finished mark has been written to the log, the transaction commit is considered finished. Objects, ODs, and the transaction finished mark can be stored in the same segment, and more than one transaction can be committed in one segment write (similar to traditional group commit). In this way, the response time can be as low as the time it takes to write one segment, and this technique should in most cases give good throughput in a single server system. However, it is difficult to implement an efficient 2-phase commit operation by using this simple technique. 2-phase commit is crucial in a multi-server system, and we have to use a more elaborate commit protocol, where more information is written to the log in the various phases of the commit process.

2-Phase Commit in Vagabond

When a transaction is started on node NodeID_c, it is allocated a transaction identifier TID_c on the node where it is started. If the transaction TID_c accesses other servers, a transaction TID_p is started on each of these servers. The transaction on NodeID_c maintains a table of all subtransactions TID_p that it controls.

At commit time, the controller (TID_c) sends a prepare command to the participating nodes. If all of the participating nodes vote yes, the controller sends the commit message to all of them. When the commit is finished they acknowledge the commit to the controller, which consider the commit finished when all acknowledge messages have been received. In more details, the commit algorithms are as follows (also illustrated in Figure 12.3):

Controller:

1. Send the `Prepare(`TID_p`,`TID_c`)` message to all participating nodes.
2. Wait for reply from all participating nodes.
3. (a) If all participating nodes vote yes, write `CommitStart(`TID_c`,timestamp)` to the log, and send `Commit(`TID_p`,timestamp)` messages to the participating nodes. When all participating nodes have acknowledged with `CommitCompleted`, a `CommitEnd(`TID_c`)` tuple is written to the log.
   (b) If not all of the participating nodes voted yes, or the controller has not received votes from all participants within the timeout period, the commit process is aborted. `Abort(`TID_c`)` is written to the log, and `Abort(`TID_p`)` messages is sent to the participating nodes that voted yes.

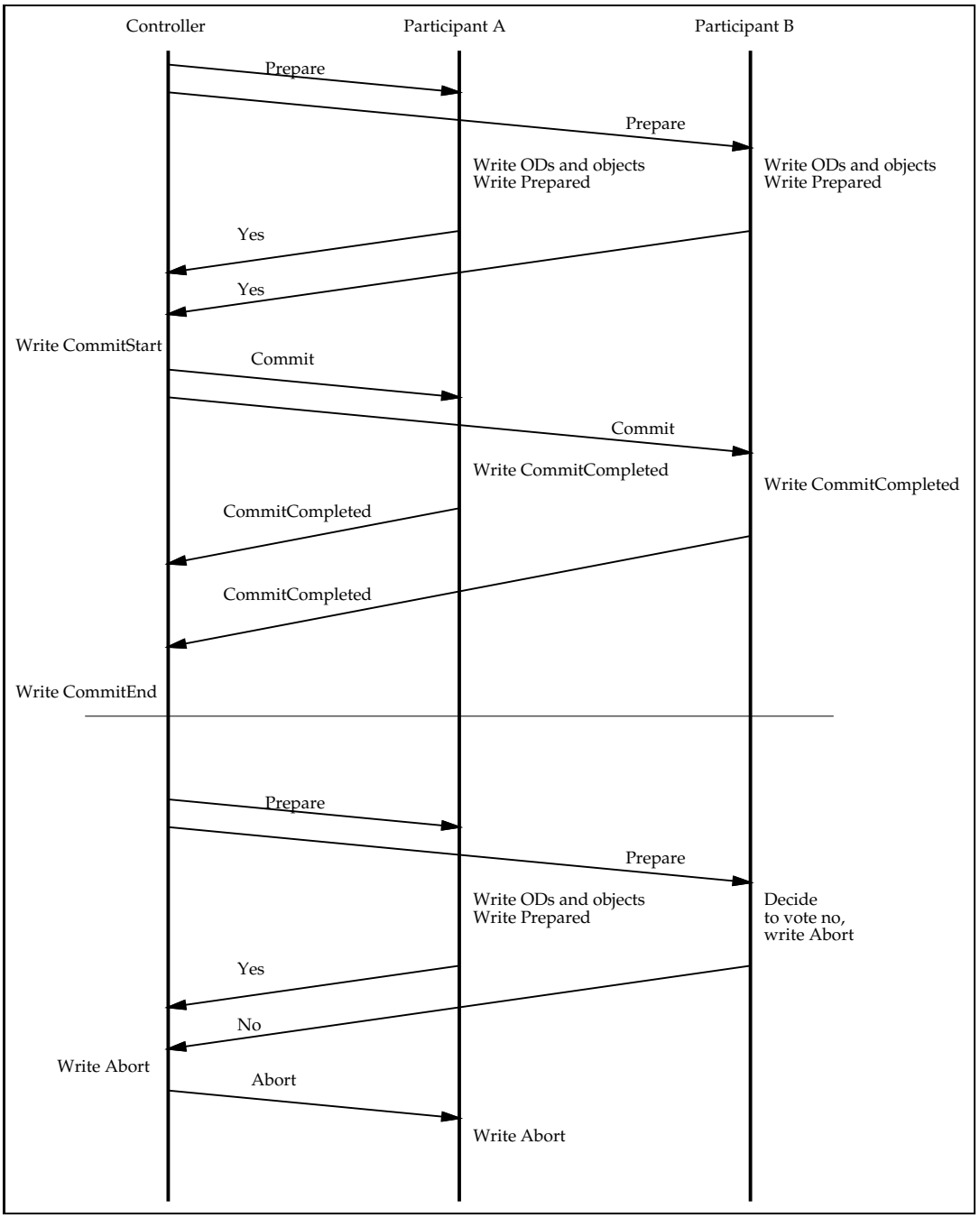


Figure 12.3: Messages and transaction information written to the log during 2-phase commit involving a controller and two participants. On the top is illustrated a commit where all participants vote yes, and the commit succeeds. On the bottom is illustrated a commit where one of the participants votes no, with an abort as the result.

Participating nodes:

1. (a) When the `Prepare` message is received and the participating node decides to vote no, `Abort(TID`$_p$`)` is written to the log, the no vote is sent to the controller, and the local transaction is aborted.

   (b) If the participating node decides to vote yes, the rest of this algorithm is executed.

2. When the `Prepare` message is received and the participating node decides to vote yes, all objects and ODs that have been created or modified by the transaction, and are still dirty in the buffer, are written to disk. In the same segment as the last objects and ODs, or in a subsequent segment,
   `Prepared(TID`$_p$`,timestamp,NodeID`$_c$`,TID`$_c$`)` is written to the log.

3. When the `Prepared` is safe on disk, a `Yes(TID`$_p$`,timestamp)` message is sent to the controller, and the participating node waits for the outcome of the voting phase from the controller.

4. (a) If a `Commit` message is received, `CommitCompleted(TID`$_p$`,timestamp)` is written to the log, and a `CommitCompleted(TID`$_p$`)` message is sent to the controller. Only after this has been done is the transaction considered committed, and created/modified ODs can be installed into the TIDX.

   (b) If an abort message is received, transaction TID$_p$ is aborted, and `Abort(TID`$_c$`)` is written to the log. It is not strictly necessary to write an abort record to the log in Vagabond, but by doing so we avoid having to ask the controller node of the outcome after a crash.

It is important to keep the local time on each node as close as possible to each other, and the commit timestamp for a transaction should be the same on each node. This is achieved by piggybacking the timestamp in the vote messages, and using these to decide the eventual commit timestamp. The commit timestamp is then piggybacked on the commit messages from the controller to the participants.

In the case of failure of one of the nodes during the 2-phase commit process, this can be handled in the same way as in traditional systems (see Bernstein et al. [12] or Gray and Reuter [77]).

If the transaction TID$_c$ has not accessed data on other nodes, the commit can by done by simply writing `CommitCompleted(TID`$_c$`,timestamp)` to the log after the dirty objects and ODs have been written, possibly in the same segment.

12.3.2 Abort

In a log-only database, we do not need to undo operations when we abort a transaction. If the transaction that wrote an object does not commit, an object written to the log before the abort operations will simply be a dead object, which will be removed the next time the segment is cleaned.

No ODs reflecting updates from a transaction will be installed into the TIDX until after the commit has been done. ODs with OIDs that have been allocated by a transaction that has aborted will never be inserted into the TIDX, and the OIDs are not reused later by any other transaction. ODs from aborted transactions that have been inserted into the PCache will be removed lazily at the same time as ODs from committed transactions are moved to the committed tree (see Section 12.2.2).

When a transaction aborts, the live-byte counts for the segments where the objects were written are decremented accordingly. This can only be done immediately for the objects where ODs are still in main memory. For those objects where the ODs have been removed from the OD cache and inserted

into the PCache, the live-byte counts are decremented when the ODs are removed from the PCache nodes.

The fact that no transaction control information is written to the log before a transaction starts the commit process, simplifies abort considerably. This can be useful in a client-server environment, and can also be used to exploit optimistic concurrency control techniques, because the commit process is only done if the validation phase succeeds.

12.3.3 Other Transaction Models

The description of transaction management has been presented in the context of ordinary flat transaction. Other transaction models could also be implemented. For example, nested transactions can be realized as a variant of 2-phase commit and reduced isolation between subtransactions.

12.4 Controlled Shutdown and Restart

A controlled shutdown of the system can be performed in several ways, where speed of shutdown is traded against the time needed to get full performance after the system has been restarted. Two alternatives are *proper* and *fast* shutdown:

1. Proper shutdown: After all ongoing transactions have finished, all objects in the object buffer are written to the log, and all dirty ODs are installed into the OIDX. The SST, PCST, and TTCT are written, and the shutdown is completed by doing a checkpoint. In this way, the system can be restarted by only reading the checkpoint information.

2. Fast shutdown: This is achieved in the same way as proper shutdown, except that dirty objects are not written back to the disk (for committed transactions, at least the delta objects have already been written to the log). This takes less time, but has one major disadvantage: when the system is restarted, object retrieval can be quite slow. For each object that is to be read, the last complete version as well as all subsequent delta objects have to be read.

There are several optimizations that can be done at the shutdown phase, that have a small cost, but could reduce the start up cost considerably. The most important is to write the most frequently used ODs. They are already resident in memory, and can be written efficiently to the log before shutdown. At startup, they can be read efficiently, thus reducing lots of costly OIDX lookups.

When the system is restarted, resident structures are rebuilt from the data written during shutdown, the most important being the SST, PCST, TTCT, the index-root node, and hot-set ODs if they have been written.

12.5 Recovery

When the system is restarted, it is determined from the checkpoint block (see Section 5.1) whether the shutdown of the system was done controlled, or caused by a crash. If caused by a crash, recovery is needed. In this section, we will take a closer look at recovery, how to handle media failures, and how to reduce the recovery time by checkpointing.

12.5.1 Crash Recovery

The purpose of crash recovery is to reconstruct a consistent state. In a traditional system, this is a very complex operation, and typically involves an analysis phase, a redo phase, and an undo phase. In a no-overwrite system like Vagabond, undo or redo of objects is not necessary. However, ODs and transaction management information is written to the log, and this information has to be read in order to rebuild the resident structures.

The first step in a recovery is to identify the last segment that was successfully written before the crash. This is done by reading the log from the last checkpoint until one of the following conditions is satisfied:

- We come to a segment that was only partially written. This means that the system crashed when it was writing this segment.
- The *next segment* of a segment does not exist. Every segment contains the address of the next segment. The address of the next segment is determined before this segment is written. If the next segment does not exist, this means that the system crashed in the interval between writing two segments.

When we read the log in order to find the end of it, we also collect all ODs, and keep each OD where we later find a commit operation for the transaction that generated that OD. For ODs that we do not find a commit, these ODs can be safely discarded because the system crashed before the transaction was committed.

After we have identified the end of the log, and processed the part of the log written after the last checkpoint, we read the log from the last checkpoint and backwards until the penultimate checkpoint. In this way, we process all segments that might have ODs from committed operation that have not yet been installed into the OIDX. This backward reading can be done efficiently, because all segments have a pointer to the previous segment. In addition, all segments also contains the number of ODs in the previous segment in the log. This makes it possible to only read the part of the segment that contains the ODs, the rest of the segment can be skipped. However, this is not necessarily beneficial. It is quite possible that many of the objects we skipped in this way, are hot-set objects that will have to be retrieved later. If this is the case, it is much cheaper simply to collect them at recovery time, to avoid random reads at a later time.

While reading the segments, we rebuild the relevant structures in memory. If we need to write index nodes during recovery because of insufficient buffer capacity, this is done to clean segments. When the log has been processed, we do a checkpoint, and when the checkpoint process is finished, the checkpoint blocks are updated. Idempotence is guaranteed because we do not modify any written data before updating the checkpoint block. If a system crashes during recovery, it will simply start recovery in the same way next time.

12.5.2 Media Failure

Media failure in a log-only system can be handled by the use of mirroring (RAID 1), RAID with parity blocks (for example RAID 4 or RAID 5), or logging to a neighbor node [93]. The use of mirroring will also improve read performance, because the read bandwidth is doubled. The write performance will stay the same.

It is interesting to note that in recent operating systems, for example Windows NT, support for mirroring is a standard feature, and can also easily be added to other operating systems, which means that media failure handling can be done outside the DBMS.

12.5.3 Checkpointing

The main purpose of checkpointing is to reduce the recovery time. This is achieved by bounding the amount of log that has to be processed at recovery time. In a traditional DBMS, the main part of the checkpoint process is to write dirty pages back to disk, usually by the use of a fuzzy checkpointing technique. In a log-only system, this is not the important issue. The log is the final repository, and objects will have to be written to the log before commit in any case. In Vagabond, the main issue of checkpointing is to install the ODs into the OIDX. This is done to avoid having to read excessive amounts of log at recovery time (in order to find ODs that have not been installed into the OIDX before the system crashed).

During a checkpoint interval, between the checkpoints, ODs from committed transactions are installed into the OIDX. To keep the installation rate high enough and reduce the amount of memory needed to store ODs not yet installed into the OIDX, all ODs from a transaction committed during one checkpoint interval, should be installed into the OIDX no later than the end of the next checkpoint interval.

The segment status table (SST), PCache Status Table (PCST), and the TID/timestamp/counter tables (TTCT), are resident in main memory, and in order to be able to recreate these after a crash, the contents of these tables are written regularly to the log. This is done by writing a certain range of entries from these tables each time we write a segment. During each checkpoint interval, all SST, PCST and TTCT entries should have been written at least once.

Checkpointing can be costly, so it is important that the amount of data to be written at checkpoint time is as low as possible, and that data structures locked as a part of the checkpointing process are locked only for a short time. In Vagabond, most operations can run as normal during checkpointing. The only restriction is that the timeout values for 2-phase commit should be smaller than one checkpoint interval. The checkpoint algorithm is as follows:

1. Wait until the number of written objects since last checkpoint, or the number of segments written since last checkpoint, reaches a certain threshold.

2. If there are ODs created before the last checkpoint that are not yet installed into the OIDX, stop all other log processing until this has been done. Note that this delay is undesirable, and can normally be avoided by giving high enough priority to OIDX updating. To reduce a possibly long checkpointing time when this situation occurs, it is possible to solve the problem temporarily (or rather postpone the problem) by simply writing to the log the dirty ODs that have not been written since last checkpoint.

3. If there are entries in the SST, PCST, or TTCT, that have not been written during this checkpoint interval, write them to the log now.

4. Update the least recently written checkpoint block. Now the checkpointing is finished, and normal operation continues.

12.6 Vacuuming

Storing an ever growing database is not always desirable, or even possible. It is therefore necessary to be able to vacuum the database, i.e., to physically delete data which has been previously logically deleted, and non-current versions of data (see Section 4.5). During vacuuming, all objects in a certain

object class or in a certain container, created before a certain time t_v are removed. Vacuuming can be done according to two different strategies:

1. Eager vacuuming.
2. Lazy vacuuming.

12.6.1 Eager Vacuuming

When eager vacuuming is used, the OIDX is searched, and the ODs of all objects that are non-current and were created before time t_v are removed from the OIDX. The segment live-counts of the segments where the objects reside are decremented accordingly. The objects themselves are physically removed during the cleaning process.

Vacuuming old versions of large objects has to be done with care, because only the modified parts of the subobject index are written when a new version is created. This means that only the parts of the large objects that are not referenced by more recent versions, can be vacuumed.

Eager vacuuming of a container is easy, but eager vacuuming of objects from a certain class is more difficult. Objects can be stored in arbitrary containers, and eager vacuuming of a class can imply traversing the whole OIDX if there is no class extent index available.

12.6.2 Lazy Vacuuming

If vacuuming is done lazily, the physical removal of an object is deferred until the segment it resides in is cleaned. An attribute in the class descriptor object (see Section 6.2.1) contains the vacuuming age t_v, so that at cleaning time a non-current object will be discarded if it is older than t_v.

With lazy vacuuming, we are not guaranteed that all objects older than t_v are inaccessible. In many cases, this is no problem, because the vacuuming is only done to reduce storage requirements. If this is considered as a problem, the age of the object can also be checked before it is used.

A problem with lazy vacuuming that is more serious, is that the live-byte count on a segment will not be decremented before the segment is cleaned. If a segments consists of objects from the same class, created at the same time, the system would never discover that it could be cleaned, it would appear to be full with live data.

12.6.3 Choosing a Vacuuming Strategy

From the discussion above, it is clear that eager vacuuming should be used when possible, i.e., when we know in which container(s) the objects to be vacuumed are stored. This will be further justified when we discuss cleaning, where we show that lazy vacuuming increases the complexity of cleaning.

12.7 Segment Cleaning

In a log-only DBMS, dead data is never overwritten. After a while, more and more of the space in the segments will contain "garbage". For example, when a non-temporal object is rewritten, the previous version becomes outdated. Without intervention, the data volume will eventually fill up, and we would not have any clean segments left. This situation is avoided by cleaning segments, i.e., to move data[4]

[4]Note that "data" in this context includes objects and subobjects, as well as index nodes and subobject-index nodes.

that is still valid from segments that have mostly dead data, and write the live data together into new segments. In this process, we get empty (clean) segments as a result.

We will denote data (objects and index nodes) in the segments as either *alive* or *dead*. Data that is alive is:

- All current versions of objects.
- Historical versions of temporal objects that are younger than the vacuuming age for the actual object class.
- Subobject and subobject-index nodes that are reachable from a large object that is still alive.
- OIDX nodes which are in the current version of the OIDX, including the PCache.

All other data can be considered dead and does not have to be rewritten to the new segment during cleaning.

Only dirty segments written before the penultimate checkpoint can be cleaned. The reason for this, is that parts of the SST, PCST, and TTCT are stored in some (or all) of these segments, and recovery processing would become more complex if these segments could be cleaned. This will not represent a problem in practice, as these "uncleanable" segments only represent a small part of the total number of segments in the log.

We will in the rest of this section first discuss how to find the state of objects and index nodes we encounter in a segment during cleaning, i.e., whether they are dead or alive. We will then discuss some cleaning heuristics, and how we, at segment creation time, can create segments that can make the cleaning less expensive. Finally, we finish this section with some concluding remarks on segment cleaning.

12.7.1 The State of the Segment Contents

One of the problems in the cleaning process is to know *in which state the objects and the index nodes in a segment are*, i.e., which are dead and can be discarded, and which are alive and need to be moved/rewritten. We will now describe how to determine the state of the objects and index nodes in a segment.

Small Objects

When an object and its OD is encountered in the log during cleaning, the object's state is found by using the following rules:

1. If the TID in the OD is the TID of an ongoing transaction, this object is alive if it has not been subsequently modified. This is determined from a lookup in the OD cache:
   (a) If the OD in the OD cache is the same[5] as in the log, the object is alive.
   (b) If the OD in the OD cache is not the same as in the log, the object is dead.

[5] In the context of these algorithms, two ODs are considered equal if they contain the same OIDs and the same physical location of the object. The timestamp attribute can be different, because the TID is used instead of timestamp in the log.

(c) If the OD is not found in the OD cache, the object as well as the OD have been written to the log due to buffer replacement. In this case, the OD has been written to the PCache, and stored in an "uncommitted tree" in a PCache node. If the PCache node is not already in the buffer, it has to be retrieved. If the OD in the PCache is the same as in the segment, the object is alive. If the OD in the PCache is not the same as in the segment, the object has been modified again, and the object in the segment is considered dead.

2. If the TID in the OD is not the TID of an ongoing transaction, this means that the object was written by a transaction that has finished. It is possible that the transaction aborted, but we do not keep a list of aborted transactions, so we can not know simply from the TID if an object in the log was created by a transaction that aborted or not. In this case, we start with a lookup in the OD cache:

   (a) If this OD is the same as in the log, *and* the object is a non-temporal object, the object is alive.

   (b) If this OD is the same as in the log, *and* the object is a temporal object,[6] *and* the object is younger than the *vacuum age* in the class descriptor object, it is alive. However, if the object is older than the *vacuum age* in the class descriptor object, it should be vacuumed. In this case the OD is deleted from the OD cache and the OIDX, and the object is considered dead.

   (c) If the OD is not found in the OD cache, we do a lookup in the OIDX:

      i. If the object is non-temporal, it is alive if the OD in the OIDX is the same as the one in the segment.

      ii. If the object is non-temporal, and an OD containing another physical location is found in the OIDX, the object has been updated. The object in the segment is outdated, and is considered dead.

      iii. If the object is non-temporal, and the OD is not found in the OIDX, the object has been deleted or the transaction that wrote it aborted. The object in the segment is in this case considered dead.

      iv. If the object is temporal, and the OD found in the OIDX, it is alive if its age is younger than the *vacuum age* in the class descriptor object. If not, it should be vacuumed, and the OD deleted from the OIDX.

      v. If the object is temporal, and the OD is not found in the OIDX, the object was either modified again by the same transaction, or the transaction aborted. In any case, the OD in the segment was never inserted into the OIDX, and the object is dead.

3. Delta objects have to be treated as a special case. If we have determined that an object in a segment is dead, and this dead object is version V_i of an object where the next version V_{i+1} of the object is a delta object, it will be impossible to reconstruct V_{i+1} if V_i is removed. The solution is to reconstruct version V_{i+1} of the object, i.e., do a read operation on version V_{i+1}, and write this version in its completed form. When this has been done, the object version V_i can be removed.

In the case an object version is found to be alive, it is rewritten to the new segment, and its OD is updated with the new location. If the OD was stored in PCache or TIDX nodes, the respective node(s) are updated.

[6] Whether an object is temporal or not, is stored in the OD, in the same segment as the object itself.

Subobjects and Subobject-Index Nodes

Subobjects and subobject-index nodes can be treated in the same way during cleaning. For simplicity, we will in the rest of this section use the term *nodes* to denote subobject-index nodes as well as subobjects.

If the node in a segment belongs to a non-temporal object, the node is alive if it is in the current version of the subobject-index tree or was written by an ongoing transaction. If written by an ongoing transaction, it can be reached from the OD created by this transaction.

If the node belongs to a temporal object, the node is alive if it belongs to an object version that is younger than the vacuum age. We do not initially know the timestamp of the transaction that created this node (see Section 12.2.3). However, we know the time when the node was first written: the initial-write time of subobjects is stored in the SOD (see Section 8.5.2), and the initial write timestamp of the subobject-index node is stored in the subject index node itself. This means that all object versions that this node is a part of, must have committed after this time. To determine if this node is alive, we check the object version which has an OD where the timestamp is the same or more recent than the write timestamp for this node. If the node is a part of this version, it is alive, if not it is dead. Note that if this version does not include the node, it is not possible that any of the more recent versions include the node, if we assume a locking protocol is used. If the node under consideration was written by an ongoing transaction, it can be reached from the OD of this object created by this transaction. If the node is alive and rewritten, the initial timestamp is kept.

As illustrated in Figure 8.13, subobject-index nodes and subobjects can belong to more than one object version, and therefore have more than one parent. This complicates the cleaning. When the node is moved during cleaning, *all* parents have to be updated. This can obviously be very costly, and temporal large objects with frequent updates should be avoided. One such example is indexes where previous versions are kept, discussed in Section 10.3.5.

As can be observed, the subobject index has to be accessed at deletion time as well as at cleaning time. It is possible that there is room for improvement in the algorithms described here, but that is left for further work.

OIDX Nodes

All OIDX (TIDX and PCache) nodes contain node identifying information, which make it easy to determine if a node in a segment is alive. OIDX nodes are not versioned, so that we do not have to take into account temporal aspects.

The disk addresses of all the PCache nodes are kept in the PCST in main memory. To find the state of a PCache node, we only need to check the entry in the PCST. If this entry points to the segment that is being cleaned, the node in the segment is alive, and should be rewritten. If not, the node in the segment is an old version, which can be discarded.

To find the state of a TIDX node, we have to traverse the TIDX to check if the TIDX node in the segment is still a node in the TIDX tree. Note that we only read the tree as far as the parent node of the node in the segment. If the node in the segment is not a child of the supposed-to-be parent node in the TIDX, the node in the segment is outdated, and can be discarded. If a TIDX node is found to be alive and is rewritten, all nodes on the path up to the TIDX root node have to be rewritten as well.

12.7.2 Cleaning Heuristics

In the cleaning process, we have several goals to aim at:

1. Free as much space as possible.
2. Cluster together related objects that are expected to be read at the same time. If we combine this clustering with prefetching, we will improve read performance.
3. Cluster together data that is expected to have the same lifetime. In this way, we avoid having to clean the same data many times.

The first goal is easy to quantify, but predicting how to achieve the other two goals is more difficult. Clustering together related objects is similar to ordinary dynamic clustering, and some of the results from related research in that area should be applicable (see Section 3.3). Clustering together data that is expected to have the same lifetime can conflict with the goal of clustering together related data, and predicting the lifetime of data can be difficult. We will now discuss heuristics that can be used to create segments that facilitate efficient cleaning at a later time.

Cleaning Conscious Segment Creation

The goal of segment creation is the same during cleaning and normal log writing: cluster together related objects, related index nodes, and in addition try to cluster data that is expected to have the same lifetime. During normal log writing, we have only limited opportunities to cluster together related objects. During cleaning, we have more objects to choose from, and can afford to spend more of the write bandwidth on achieving good clustering.

A single segment can contain data from all the categories mentioned, i.e., PCache nodes, TIDX nodes, and objects. However, the expected lifetime for these categories are very different:

1. PCache nodes in particular get invalidated fast. The size of the PCache will usually be relatively small, compared to the TIDX, and the nodes are uniformly accessed. This results in a short average lifetime for the PCache nodes.
2. The TIDX is also continuously rewritten, and each time a node is rewritten, the previous version becomes dead. Nodes in the upper levels of the tree are very frequently rewritten, but nodes from the lower levels in the tree are less frequently updated.
3. Temporal object versions are typically long-lived, they are mostly removed as a result of vacuuming. Non-temporal object versions exist in two categories: some of the objects are very rarely updated, while others are quite frequently updated, invalidating the previous versions.

We can use this knowledge when segments are created. To increase efficiency and reduce the cleaning costs, we write the PCache nodes, TIDX nodes, and objects to separate segments when possible. Segments storing TIDX and PCache nodes in particular will very soon get a live-byte count close to zero, which makes these segments excellent candidates for cleaning. Cleaning of these segments is cheap: there is little data that needs to be rewritten, and the state of the nodes can also be found at a low cost. Note that OIDX operations are done asynchronously with the other operations, so that we do not increase the transaction response time when we try to collect TIDX nodes and PCache nodes in separate segments.

It is very difficult to predict the expected lifetimes of objects, and it is better to try to optimize on good clustering rather than doing uncertain predictions. However, if the load is high enough, it will be beneficial to write temporal and non-temporal objects to separate segments, because we expect these to have different lifetimes. If the load is not high enough, this approach will result in longer response

times, or in writing of partial segments. Partial segments should be avoided when possible, because the reduced write size increases the relative write cost.

12.7.3 Concluding Remarks on Segment Cleaning

The cleaning algorithms we have outlined may appear complex and costly. However, the situation will normally not be as bad as it might appear. Much of the complexity and associated cost comes as a result of lazy vacuuming, and this will not affect the cost of cleaning objects from object classes where lazy cleaning is not used.

Avoid Lazy Vacuuming of Large Objects. Lazy vacuuming on large objects will in particular result in expensive cleaning. The solution is to avoid lazy vacuuming on large objects by storing the large objects in containers, and instead use eager vacuuming on these containers.

Cleaning Object Segments. Cleaning can be costly if we have to clean segments which contain many objects. For each object we move, the physical location in its OD has to be changed. By separating temporal and non-temporal objects, much of this problem should be solved.

Batch Cleaning. To reduce the cleaning cost, it is beneficial to clean as many objects as possible at a time. By doing that, it is easier to create good clustering when rewriting objects, and the average lookup costs when determining the state of the data is reduced.

Hole-Plugging. With high disk utilization, and little idle time, segments do not have time enough to let its data get "dead". In this case, the cleaner will have to process many nearly full segments to get enough free space for one new segment. This can be relatively costly. In LFS, this dangerous point is reached at about 80% disk utilization. In LFS, it has been shown that the cleaning cost at high disk utilization can be reduced by the use of *hole-plugging* [144], instead of traditional cleaning.

When doing hole-plugging, one segment is read, and its contents are written into empty holes in existing segments. Atomicy is preserved by the fact that only whole disk blocks are written into the segments, thus, a crash during update will not destroy any data that is alive. The hole-plugging is followed by writing the updated metadata to the log. This approach works well in a system using fixed-sized blocks as the storage granularity, but it is more difficult to do it this way when variable sized blocks are used. In Vagabond, we also have the segment header that needs to be updated. For this to be done safely, we would have to write the existing segment header to the log before we update it. Such operations will cost several disk seeks, and will complicate the recovery operation considerably, because we will need to employ several recovery strategies after a system crash. Additionally, we do not get the benefits of dynamic reclustering, as is possible during normal cleaning. For these reasons, we do not consider hole-plugging as a suitable alternative in Vagabond.

Bitmaps for OIDX Node State. A significant cost when cleaning segments that contain OIDX nodes, comes from the OIDX lookup which is done in order to determine whether an OIDX node is valid. This cost can be reduced by maintaining a bitmap for every segment containing OIDX nodes, using one bit for every OIDX node. The bit is set if the OIDX node is valid, and when an operation that results in invalidation of an OIDX node (update, split or delete) is performed, the appropriate bit in the bitmap is reset. In order to know which bit to reset, we keep the segment and slot number of all main-memory resident OIDX nodes in main memory. The bitmap is checkpointed to disk a regular

times, and can even be segmented if it is large. When a segment is cleaned, we do not have to do an OIDX lookup for the nodes where the valid bit is reset.

12.8 Schema Management

Schema management, for example schema creation, removal and update, is easy to support, because classes are defined by class descriptor objects, which can be versioned. The schema itself can also be stored as an object.

12.8.1 Creating Classes

Creating a new class is done by creating a new class descriptor object (see Section 6.2.1), which is stored like an ordinary object.

12.8.2 Removing Classes

Removing a class is done by deleting the class descriptor object (CDO). The CDO is a temporal object, which means that even if a class is removed, the class and its objects will still be accessible for temporal accesses. If we want to physically remove the objects in the class, this has to be done through vacuuming, with vacuuming age *now*, i.e., delete all objects of the class (see Section 12.6). Removing a class has the same consequences as removing containers (see Section 8.4.2), and has to be done with care.

12.8.3 Class Modification

The class metadata is stored in a class descriptor object (see Section 6.2.1), and class modification is done by updating the class object of the actual class.

Adding attributes to a class increases the size of the objects in that class. In traditional systems, where in-place updating is used, this situation can be solved by rewriting of all the objects in the class to a new place, with the new size. This is done either in a lazy or eager way. In a log-only system, class modification can be handled in the same way. When in-place updating is used, clustering (and performance) will be affected by a class modification operation, but this is less of an issue in a log-only system.

12.8.4 Object Class Migration

In some languages, an object can be migrated from one class to another. In Vagabond, this is done simply by changing the class identifier in the OD. In addition, the object may or may not be converted. This has to be handled by the application language.

12.9 Object Migration

As a part of a distributed system, object migration should be supported. In Vagabond, object migration from one server group to another is supported by storing forward information in the OIDX of the server where the object was created (see Section 8.8). If the object is migrated a second time, only the index entry on the server where it was created will be updated. In this way, only one indirection is needed. By caching remote ODs on a server, this will only infrequently require network traffic.

12.10 Backup

As with other DBMSs, backup could be performed by copying the contents of the data volume to a backup medium. In addition, backup could easily be done on-line. Because the written segments are timestamped and a no overwrite strategy is used, there will be no consistency problems even if the system is running.

Incremental backup is also possible, it is enough to know the last time of backup to know where backup should be started. The incremental backup could also be done on-line, and again, even if we stop backup when the load is high, we know where to continue in less busy periods.

If incremental- or on-line backup is done, it is important that the backup rate is high enough to keep pace with the segment creation rate.

12.11 Query Processing

Queries are essentially lookup operations, either on single objects, or on a set of objects. Query processing in Vagabond is similar to the way it is done in other systems.

Intermediate results from complex queries, as well as temporary results from, for example, hash-based partitioning algorithms can all be stored in *temporary segments*. These are segments used only to store temporary data. Doing it this way has several advantages over how it is done in existing systems. There is no need to reserve disk space for query processing. Segments are allocated when needed, and deallocated when they are not needed anymore. This also means we do not have to reserve certain disks for query processing. As a result, available disks and disk bandwidth can always be used in the most efficient way.

12.11.1 Non-Temporal Queries

Non-temporal queries are traditional queries, where only the current version of objects are involved. For such queries, traditional algorithms can be used. This includes hash-based partitioning [29, 30] as well as extensions for queries involving methods and path expressions.

12.11.2 Temporal Queries

Many simple temporal operations can be satisfied from OIDX lookup operations, discussed in Section 8.4.2. More complex queries can utilize existing algorithms for temporal query processing (see Section 4.3 for an overview of typical temporal query operations, and Section 4.6.4 for references to temporal query algorithms).

12.12 Volume Management

The log is stored in one volume, which consists of one or more storage devices (see Section 6.1.5). To have a known entry into the log, one of the devices functions as the *root device*. Volume information and checkpoint areas are stored on the root device. The root device is typically the fastest device in the system, to keep checkpointing as efficient as possible.

All addressing in the log is done using logical addresses. For each device, a part of the logical-address space is allocated, according to its size. In this way, addressing is uniform. In the case of removable media, for example optical disk and tape, the device needs to get allocated enough space from the logical address space for each of the tapes or disks that will be used.

Device size, characteristics, and other important information, is stored in a *volume device table* (VDT). This table is also used to do the mapping from logical address to device and physical address. The physical data structures will be described in more detail in Chapter 13.

12.12.1 Volume Formatting

Before the storage manager can use a volume, the volume must be formatted, i.e., made ready for use by the system. Before a volume is formatted, at least one device must be available for the volume to function as the *root device*.

The segment size for this device has to be decided. Based on the total device size, the segment size is used to calculate the number of segments. The initial device information block and the initial checkpoint blocks are initialized and written to the root device.

12.12.2 Adding and Removing Devices

Adding a new device can be done simply by initializing and writing the device information block to the new device, and updating the volume device table.

Removing a device is also easy, but before it is removed, the data residing on the device has to be moved to another device. This is done by cleaning all segments on the device, and rewriting the data that is still alive to another device. After this has been done, the device is removed from the VDT.

12.13 Transparent Use of Tertiary Storage

Even if disk space is cheap, it will still be necessary to have data on tertiary storage for some applications. This is only partially supported in current ODBMSs. Most of the systems are only able to use tertiary storage as a backup medium, and do not use it as an integral part of the storage system. We are only aware of one system that can use tertiary storage in a more transparent way, the Objectivity ODBMS. This is achieved by the use of the *Objectivity Open File System* as a middle layer between the ODBMS and the storage system [167]. In Objectivity, transparent use of tertiary storage is achieved by interfacing to the *High Performance Storage System* (HPSS). The HPSS supports storage hierarchies, and files can be moved up and down the hierarchy based on usage patterns, storage availability, and site policies.

Vagabond could also be designed to interface to the HPSS, but this introduces another component in the total system, increasing price and complexity. For most applications area, support directly from Vagabond is preferable. This also gives the DBMS more control on where data and indexes are in the storage hierarchy, which can improve the performance. In Vagabond, references to physical locations in the log are based on uniform addressing, which means that segments can be written to disk and tertiary storage in the same way. Migration to tertiary storage can be done at cleaning time, based on usage statistics, it can be done as a part of the vacuuming, or it can be done explicitly by moving whole containers to tertiary storage.

12.14 Node Operations

Objects are declustered over the servers participating in a server group. When adding a server to a group, the declustering strategy has to be changed so that objects are stored on the new node as well. Without an explicit reclustering, only new objects will be placed on the new node. Removal of a

node is possible, but reclustering (moving data from the node, which can be very time consuming) is necessary before it can leave the configuration.

The number of servers in a server group can change with time, which implies that the declustering function will change with time. Declustering is done on container or class granularity, and in order to be able to retrieve objects, a table for each container or class is necessary to track of the number of nodes participating in the server group during a certain interval (interval based on OID creation).

12.15 Summary

This chapter described the most important operations in Vagabond. In the next chapter, we will describe in more detail the physical data structures that these operations manipulate.

Chapter 13

Physical Data Structures

In this chapter we describe the most important physical data structures in Vagabond. First, we describe the data structures used in the data volume (the log), and then we describe the most important main-memory data structures.

13.1 Data Volume Structures

The log is stored on a data volume, which consists of one or more devices (see Section 6.1.5).[1] Figure 13.1 illustrates the layout of data structures on the devices in a data volume:

1. A device information block on each device in the data volume.

2. Checkpoint blocks (only on the root device).

3. A number of segments.

4. A volume device table, describing the devices in the data volume (only on the root device).

One of the devices in the data volume is called the *root device*, and is the entry point for the data volume. The volume information block and the checkpoint blocks are always on this device. The checkpoint blocks are at a known location on the root device, so that we are able to do restart based on the information in these blocks.

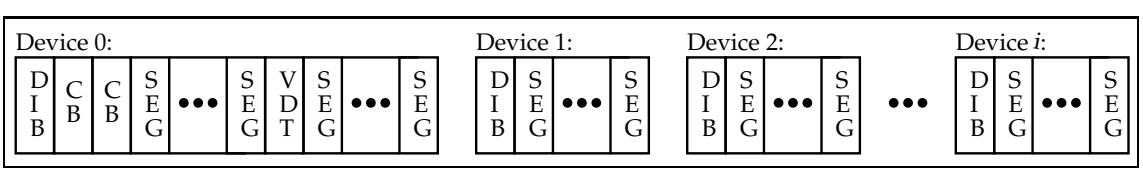


Figure 13.1: A data volume, consisting of i devices, where device 0 is the root device. DIB denotes a device information blocks, VDT denotes a volume device table, CB denotes a checkpoint block, and SEG denotes a segment.

Timestamp
Storage manager version
Segment size
Number of segments
Access characteristics
Write characteristics
Removable?
Timestamp

Table 13.1: Device information block.

13.1.1 Device Information Blocks

The device information blocks (DIB) hold static device information that is normally only read when the system is started. They are written when the device is added to the data volume, and are always stored in the first addressable location on the device.

The most important contents of the DIBs are summarized in Table 13.1. Segments on different devices can have different sizes, but all segments on one device have the same size. The *segment size* is the segment size used on this device. The *access characteristics* in the device information block contains information to use in cost function based optimization, for example average access time and data transfer bandwidth. *Write characteristics* contains information on how many times a block on the device can be written to. For example, on a magnetic disk we can write to the same physical sector many times. On other devices, the number of writes to one physical location is more restricted. For example the number of physical writes to a block on flash memory is restricted, and for some types of optical storage we can only write once to the same location. *Removable* is set to true if this is a removable medium.

13.1.2 Checkpoint Blocks

The checkpoint blocks contain the information necessary to restart a system. If the system stopped because of a crash, the checkpoint blocks are used to know where to start recovery (see Section 12.5).

There are at least two checkpoint blocks. When doing a checkpoint, the new checkpoint information is written to the block least recently updated (ping-pong strategy). The blocks contain timestamps, so that during recovery, the last successfully written block will be used. If the system crashes during the update of one of the blocks, the two timestamps in the block will differ, and we use the other one. The checkpoint blocks will always be stored immediately after the device information block on the root device. In this way, they are at a known location.

The design described here has only one checkpoint region with two or more checkpoint blocks. A single checkpoint region has the unfortunate disadvantage of giving a single point of failure, a media failure might easy destroy all the checkpoint blocks. To increase reliability, several checkpoint regions could be used.

The checkpoint block information is summarized in Table 13.2. If the DBMS shutdown was controlled, *shutdown checkpointing* is true. In this case, no recovery is necessary.

During a controlled shutdown, the TIDX root node and the contents of the SST, PCST, and TTCT are written to the log. Although this information is written into ordinary segments, we call these

[1] Actually, each device needs to be replicated in order to be able to handle media failure.

Timestamp
Location of first segment written during this checkpoint interval
Location of last segment written during this checkpoint interval
Location of first shutdown segment
Location of VDT table
Shutdown checkpointing?
Timestamp

Table 13.2: Checkpoint block.

Timestamp
Storage manager version
Total number of segments
Number of devices
A copy of the DIB contents for each device
Next segment in VDT
Timestamp

Table 13.3: Volume device table.

last segments *shutdown segments*. These segments are read when the system is restarted in order to initialize the SST, PCST, and TTCT in main memory.

13.1.3 Volume Device Table

The volume device table (VDT) contains information on the devices in the data volume (see Section 12.12), and is stored in one or more segments. There is no restriction on the size of the VDT, and it can span several segments. It is always stored in segments on the root device, because it is needed before the other devices can be accessed. It is read when the system is started, and written when a data volume is created, and when new devices are added to or removed from the data volume. Its most important contents are summarized in Table 13.3.

13.1.4 Segment Structure

There are three types of segments in Vagabond:

1. Ordinary segments, which contain objects, OIDX nodes, transaction control information, and persistent copies of the SST, PCST, and TTCT.

2. Temporary segments used to store intermediate results during query processing.

3. Segments storing the VDT.

All segments consist of a segment header, followed by segment type specific data. Temporary segments are only used as "scratch segments", and the data stored in these segments are specific for the operation creating the temporary segment. VDT segments contains some or all of the entries in the

VDT. In the rest of this section, we will describe the contents of ordinary segments, summarized in Table 13.4.

Segment Header. The segment header consists of a timestamp, pointers to the previous and the next segment, and a checksum. As noted by Seltzer [185], there is a problem related to the atomicy of segment writing. In a traditional system where less than one track is written at a time, we can know for sure if a block was successfully written in its entirety by comparing the timestamps at the start and at the end of the block. When we write a segment, we write multiple tracks, and can not know if the whole segment has been written only by comparing the start and end timestamps. The disk controller often tries to optimize writes, and might write the blocks of the segment in a different order. Therefore, a checksum is computed over the blocks in the segment, and is included in the segment header.

To speed up recovery, every segment contains the address of the previous and next segment in the log (the address of the next segment is determined before the current segment is written).

Even with an adequate segment size, there might at times be so little write traffic that to keep commit times low, we need to write a segment even if we have very little dirty data. In this case, we follow a strategy similar to that used in BSD-LFS [185], and write partial segments, here called *subsegments*. When writing subsegments, we effectively write several (small) segments into one of the ordinary physical segments. This has no effect on the description here, the only difference is that the *next segment* field now might point to a subsegment in the same segment, and that write efficiency can be lower than when using full size segments.

Object Related Information. The main part of the segment consists of objects, OIDX nodes, and transaction control information. There are some details to note here. When a segment is to be cleaned later, it is necessary to know the contents of the segment, in order to be able to determine which objects and OIDX nodes are still valid. If we did not store this information in the segment, we would for example have to search through the whole OIDX to find out which objects resided in the segment. This is obviously too costly. The solution is to always store the OD in the same segment as the object. The object and the OD can be stored before the transaction commits, before the transaction commit time is known. Because of this, ODs stored in the log always contain transaction identifiers instead of timestamps. When a transaction commits, the transaction ID and timestamp is written to the log, and this can be used during recovery if needed. By doing it this way, objects and ODs can be written before a transaction commits because the buffer is full, or simply to avoid a heavy burst at commit time. Having the OD available is also useful if we do eager prefetching, which can be done by reading the whole segment, even if only a part of it is needed. Given the OD, we only need a search of the available ODs in memory, and maybe the OIDX, to have all the necessary information available.

OIDX and subobject index nodes contain enough information inside the nodes to be able to identify them, so they do not need any additional identifying information.

Transaction Control Information. Transaction control information is written to the segment as would be done in an ordinary write-ahead log, but *no* information is written until the transaction starts the commit process (see Section 12.3.1). The TID in the ODs of written objects will only be needed for objects whose transactions have committed, so it is not necessary to record the transaction ID before the transaction process starts.

When a transaction prepare starts, the transaction identifier and timestamp is written to the log. When the transaction finishes its commit, or aborts, this information is written to the log. In general,

Timestamp
Physical address of previous segment
Physical address of next segment
Segment checksum
The number of TIDX nodes
The number of PCache nodes
Number of ODs in previous segment
Number of ODs
Number of small objects
Number of large object subobject index nodes
Number of SODs
Number of large object subobjects
Number of transaction `CommitStart`
Number of transaction `CommitEnd`
Number of transaction `Abort`
Number of transaction `Prepared`
Number of transaction `CommitCompleted`
Range of entries from SST (start segment, # of entries)
Range of entries from PCST (start node, # of entries)
Number of TTCT entries
TIDX nodes
PCache nodes
ODs
Small objects
Large object subobject index nodes
SODs
Large object subobjects
$(\mathtt{TID}_c, \mathtt{timestamp})$ for each `CommitStart`
$\mathtt{TID}_c$ for each `CommitEnd`
`TID` for each `Abort`
$(\mathtt{TID}_p, \mathtt{timestamp}, \mathtt{NodeID}_c, \mathtt{TID}_c)$ for each `Prepared`
$(\mathtt{TID}, \mathtt{timestamp})$ for each `CommitCompleted`
SST entries
PCST entries
TTCT entries
Timestamp (aligned to end of last block)

Table 13.4: Segment structure.

the prepare and commit/abort finished will be written in different segments.

Persistent Copies of Main-Memory Tables. Persistent copies of the SST, PCST and TTCT are also stored in the segments. The range of SST entries is indicated in the SST range field, which indicates the range as the start segment and number of entries. This is similar for the PCST, where the range is a range of nodes. In the case of TTCT, the entries are self describing, so that only the number of entries is necessary. A zero value in number of entries for any of these tables means that there are no SST/PCST/TTCT entries in this segment.

13.2 Memory Data Structures

In this section we describe the most important memory structures in Vagabond, as illustrated in Figure 13.2:

- Object descriptor cache (OD cache).
- Small object buffer.
- Large granularity buffers (Subobjects and PCache-, TIDX-, and subobject index nodes).
- SST buffer.

The implementation of the PCST and the TTCT is straightforward, and we do not discuss these any further in this chapter.

The size of the OD cache and the buffers can be adaptively resized. The cost equations derived later in this thesis and in the papers in the appendixes can be used to find optimal sizes for the different buffers with changing access patterns.

13.2.1 Object Descriptor Cache

As described in Section 8.6.1, an OD cache is used to reduce the OIDX access costs, and reduces lookup costs as well as index update costs. ODs retrieved from lookups in the OIDX are inserted into the OD cache when retrieved, and new ODs resulting from object updates are inserted into the OD cache. However, new ODs from new objects are not initially inserted into the OD cache, they are written directly to the OIDX.

Designing an efficient OD cache is not straightforward. The requirements and functionality of the OD cache require a careful design. The entries in the OD cache are of a fine granularity, which means that additional overhead data can have a larger impact on performance than it would have in a page buffer, where the additional overhead usually is very small compared with the buffered items themselves. We will now describe operations that the OD cache has to support, study some aspects of the writeback of ODs to the OIDX, and then describe the architecture of the OD cache.

OD Cache Operations

The OD cache has to support the following operations:

- `lookup_current(OID)` Returns the OD of the current version of the object if the OD is in the OD cache.

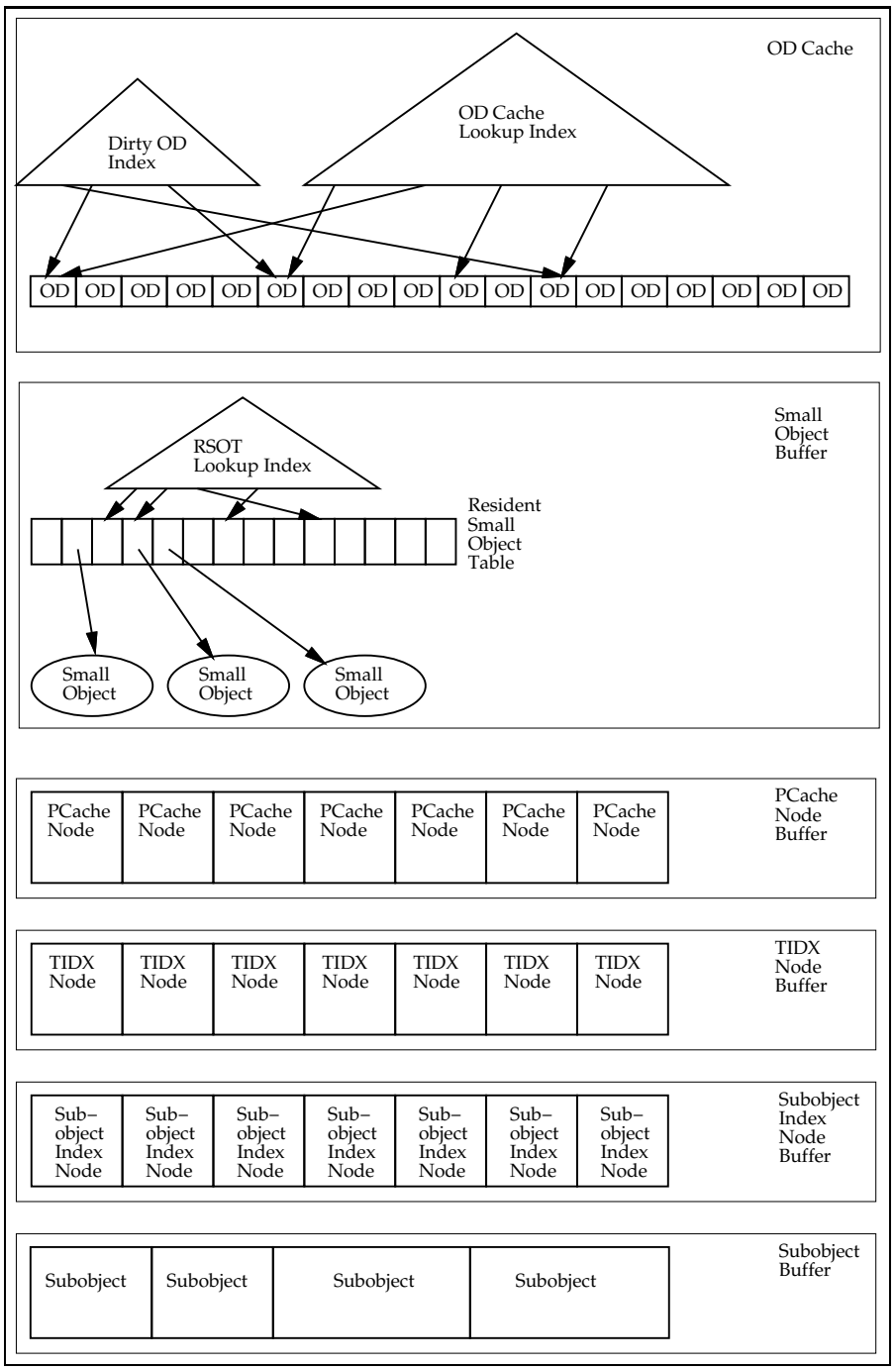


Figure 13.2: Important memory buffers. Other important structures include the segment status table (see Chapter 5.1) and the PCache status table (see Chapter 9.3).

- `lookup_most_recent(OID)` Returns the most recent OD with a particular OID resident in the OD cache.
- `lookup_at(OID,TIME)` Returns the OD of the object version valid at `TIME` if the OD is in the OD cache.
- `lookup_start(OID,TIME)` Returns the OD of the object version that has start time (commit time) `TIME` if the OD is in the OD cache.
- `lookup_end(OID,TIME)` Returns the OD of the object version that has an end time `TIME` if the OD is in the OD cache. The end time is equal to the start time of the next version of the object (or delete item if this was the last version before the object was deleted).
- `insert(OD)` Inserts OD into the OD cache. If this is a new current version of an object, set the end timestamp of the current version of the object if resident in the OD cache.
- `remove(OD)` Removes an OD from the OD cache.

In order to iterate through versions of an object, a combination of a `lookup_at()` and subsequent `lookup_start()` operations can be used.

OD Cache-to-OIDX Writeback

Because of the size of each item in the OD cache, it is very important that when dirty[2] ODs are to be written back to the OIDX, this can be done in batch. To reduce the amount of data that has to be read (installation read of OIDX nodes) and written, dirty ODs are sorted so that ODs that belong to the same OIDX nodes can be installed into the OIDX nodes at the same time. In most cases, disk seek time will also be reduced by updating the OIDX from the list of sorted ODs. This is similar to general use of an elevator algorithm when writing back pages from a page buffer.

A complicating factor for the OD cache, is the potentially large number of entries that have to be sorted. This can be time consuming. To have a sorted list of ODs, two approaches can be used:

1. When all dirty ODs from the last dirty list have been written back to the OIDX, a new list is generated by creating an array with pointers to all dirty ODs. This array is sorted, based on the OIDs, and then the ODs are asynchronously written back. The advantage with this approach, is that the extra space overhead is minimal. However, this approach has two important drawbacks:
   (a) ODs created after the array has been created and sorted, will have to wait until the next checkpoint interval, even if they belong to one of the OIDX nodes that is retrieved and written when the array is processed.
   (b) The sorting of ODs can take several seconds of CPU time.
2. A dirty OD index with ordered elements, for example a binary tree, can be used. When a new OD is created, a pointer to the new OD is inserted into the index. During each checkpoint interval, the index is processed at least once. Because new entries are inserted immediately, we avoid the problem with the previous approach, where only ODs created during the previous checkpoint interval were available for the OD cache-to-OIDX writeback process. The disadvantage

[2]Note that *dirty* in this context means *dirty with respect to the OIDX*, i.e., new or a modified ODs that have not yet been inserted into the OIDX. Persistent copies of these ODs have already been written together with the objects to the log.

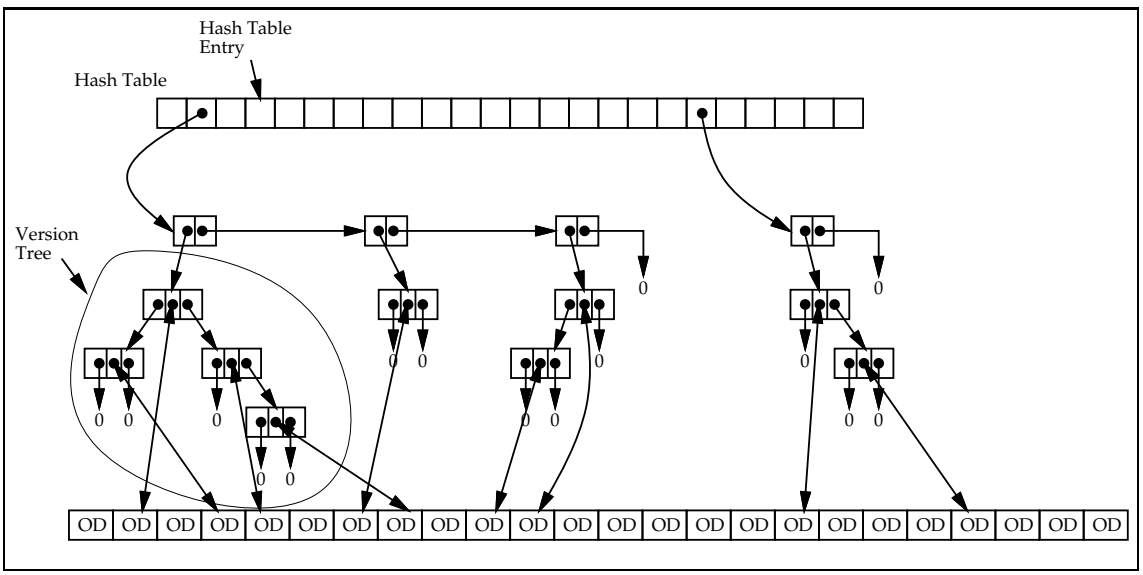


Figure 13.3: The OD cache lookup index.

of this approach is a higher space overhead. For example, using a binary tree, two pointers are needed for each dirty OD. Note that a general priority queue is not sufficient for this index. The reason is that at some point in time, we have to be guaranteed that all entries inserted before a certain time (in this case, before the previous checkpoint interval), have been processed (in this case, before we can finish a new checkpoint).

We expect the space overhead of the dirty entry index to be compensated by increased writeback efficiency, and choose the dirty entry index approach.

OD Cache Architecture

It is possible to store *all* ODs, dirty as well as clean, in the index tree, and use the index tree as the access path for OD cache lookups as well. However, if that approach was used, we would have to scan through the whole index tree during each checkpoint interval. If most of the entries are clean, many CPU cycles will on average be needed in order to find the dirty ODs.

A better approach is to use one lookup index for all ODs in the OD cache, in addition to the dirty OD index. The lookup index is optimized for accesses to the OD of the most recent version of an object. The dirty entries are also indexed by this index, which means they are represented in both the dirty entry index and the lookup index. The reason for this, is that without this redundancy, we would in many cases have to search both indexes when doing a lookup for a recent OD.

OD Cache Lookup Index

To understand the design of the OD cache index, it is important to remember that each update of an object generates a new OD. For each object, there will be one OD for each version, and more than one of these ODs can be in the OD cache at the same time. This means that even though the most frequent lookup operation is to retrieve the most recent OD of an object, it must be possible to store the other ODs in the OD cache as well, and it must be possible to retrieve these in an efficient way.

The index is based on a chained overflow hash table. The bucket to put an OD into, is chosen based on hashing the OID of the OD. In this way, all ODs of the same object (same OID) will be in the same bucket. The ODs of an object are inserted into a version tree, for example a binary tree, where time is used as the key. ODs with different OIDs can be hashed to the same bucket, and for each OID we have a separate version tree. The version trees are chained in a linear list. With an appropriate size of the hash table, the number of OIDs hashed to the same bucket should be low.

The architecture of the OD cache lookup index is illustrated in Figure 13.3. Each entry in the hash table is a pointer to a list with pointers to the version trees. As can be observed, it would be possible to include the pointer to the next binary tree in the root of each version tree. In this way, we would avoid one pointer dereference. However, this is not done, because it could make some tree operations more complicated.

When choosing an appropriate version tree, the most important goals to achieve are 1) low insert cost, especially of a new current version OD, and 2) low lookup cost for the current version OD, which will be the most frequent operation. An ordinary binary tree is one possible solution. However, a problem with storing the ODs in a binary tree, is that if entries to be inserted into the tree have monotonically increasing key values, the result will be a linked list. Unfortunately, this is exactly the case when the inserts into the OD cache is ODs of new versions: The key value TIME is constantly increasing. One solution to this problem is to use a balanced tree, for example a splay-tree or a 2-3-tree. However, this increases the insert and space cost,[3] and it is not certain that this approach will reduce the *average* access cost. Based on the knowledge of insert pattern and average number of versions, other heuristics can perform better, for example:

- When a new current version OD is inserted, its node is made the new root of the tree, and the current version of the tree is made the left subtree of this node. Non-current ODs are inserted into the tree following the binary tree insert algorithm. With this approach, searching for the current version has a low cost.

- Another option is to keep a counter c which is increased for every insert of a new OD into the tree, and decreased for every delete from the tree (but always non-negative, i.e., if c is zero and we have a deletion, c will remain zero). If c reaches a certain threshold, the tree is reorganized and c is set to zero. Although a reorganizing approach in general is a bad idea, with higher cost than using a balanced tree, it can perform well in the OD cache if we assume that only a very few of the version trees have an insert rate that is high enough to result in reorganizations. The space overhead is low for this approach, as we only need an additional counter for each version tree.

- On average, it is even possible that a list could perform well. The problem with this approach, is the high worst case cost.[4]

OD Cache Replacement

The OD cache will only have empty slots during startup, before enough objects have been accessed to fill up the OD cache. After the cache has filled up, one of the ODs resident in the OD cache has to be discarded before a new OD can be inserted. Only non-dirty ODs (with respect to the OIDX)

[3] It is possible to implement the splay tree with the same space cost as a binary tree, but this increases the access cost.

[4] Rastogi et al. used a linear list for versions of data items in the Dalí main-memory storage manager [176]. However, in Dalí, versioning is only used to support transient versioning, and not to provide support for temporal data. Hence, the length of a version list will usually be short.

OD
Memory address
Uncompressed object size
Status (clean or dirty)

Table 13.5: Entry in the resident small object table.

can be discarded, and the clock algorithm (see Section 3.7) is used to decide which of the candidate ODs should be discarded. The number of dirty ODs in the OD cache should be kept relatively low, to reduce the cost when searching for a candidate OD for replacement.

13.2.2 Small Object Buffer

In Vagabond, small objects are stored and retrieved as separate entities, and an object buffer (see Section 3.6.2), is the only reasonable choice. Large objects have to be treated differently, because some of the subobject index nodes and the subobjects of an object version might also be a part of other object versions. In this section aspects of the small object buffer are described, and in the next section buffering of large object subobject index nodes and subobjects are described.

Modified Object Chain

For each active transaction, there is a *modified object chain*. This list contains the objects that have not yet been written to persistent storage, but must be written before the commit operation (see Section 12.3.1) can finish.

Small Object Buffer Architecture

For the objects in the small object buffer, a clock algorithm is used as an LRU approximation. The clock algorithm, as well as buffer space allocation with variable size granules are described in Section 3.7.

The *resident small object table* (RSOT) is used to store administrative information on objects currently resident in the small object buffer. The access to the RSOT is through an index structure similar to the one used for lookups in the OD cache.

Although the information stored in the RSOT alternatively could be stored together with the ODs in the OD cache, the number of objects resident in memory is in general much smaller than the number of entries in the OD cache, making that approach less space efficient.

When an object is read into the buffer, its OD is removed from the OD cache, and reinserted into the OD cache when the object is discarded from the object buffer. Although this at first glance might seem to be inefficient, it simplifies the OD cache management considerably, and also has the benefit of removing interaction and synchronization between the OD cache and the small object buffer.

Table 13.5 summarizes the contents of a RSOT entry. When a small object is retrieved, the memory location and the size of the object is inserted into the RSOT, together with the memory location and the size of the object. The reason for storing the object size in the RSOT entry, is that the object size in the OD is the size of the object while on persistent storage. On persistent storage the object might be compressed, and thus have a size different from the size when in main memory. Using the physical location field in the OD to store the main-memory location of the object could be done to

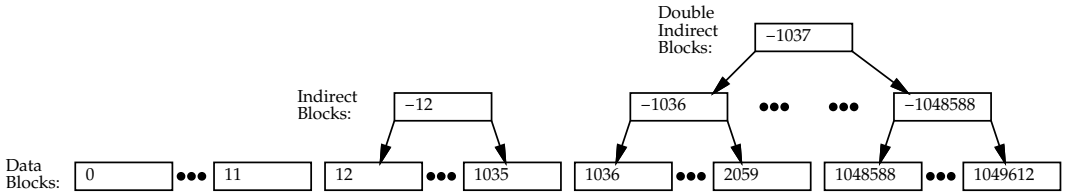


Figure 13.4: Block numbering in BSD-LFS [183].

save space, but if that was done, we would have a problem when the object was discarded from the buffer. We would then have lost the log address, and the OD would have to be discarded as well. That would make it necessary to do a costly OIDX lookup the next time the object was to be accessed.

Note that when an object is updated, a new OD is created for the object. If both versions are to be stored in main memory, a new RSOT entry has to be created for the new version.

13.2.3 Large Granularity Buffers

The most frequently used subobjects and TIDX-, PCache-, and subobject index nodes, hereafter simply called items, are buffered in the large granularity buffers. In Figure 13.2, separate buffers are used for each of the categories, but it is possible to use a common buffer if desired.

Large Granularity Buffer Indexing

In traditional disk page buffers, buffer items are indexed on disk addresses, and accessed through a hash table. This is not straightforward in the case of tree-based indexes in a log-only system. We know the addresses of items already written to the log, but because these addresses are only available after writing, we have a problem with new items. In LFS systems, this problem can be solved by numbering the nodes in the index, one such ordering is described in by Seltzer et al. [183], as illustrated in Figure 13.4. The leaf nodes, containing the data, are numbered with zero and positive numbers, and indirect blocks are numbered with negative numbers. However, this simple ordering is difficult to extend to the case of hierarchical indexes (the TIDX) and overlapping trees (subobject indexes). Our solution is to index on log address for those items that have been assigned a log address. Items that are not yet assigned a log address, are indexed by memory address.

Figure 13.5 illustrates the buffer indexing. In the figure, we assume that we already have the OD of object i resident, together with parts of the subobject index. Some of the subobjects may also be resident, but are not shown in the figure. At time $t+j$, a new version of the large object is created. The newly created or modified subobjects have to be written to the log before a transaction can commit, but the subobject index itself can be written asynchronously to the log at a later time during the same or following checkpoint period. In the case of a system crash, the existence of the subobjects in the log can be use to recreate the dirty subobject index that was only resident in main memory at the time of the crash. Until the subobject index nodes are written back, memory pointers are used as references, as illustrated in the figure. At time $t + k$, the large object is modified once more, and again, the subobjects have to be written to the log before the transaction can commit, but the subobject index itself can be written asynchronously to disk at a later time.

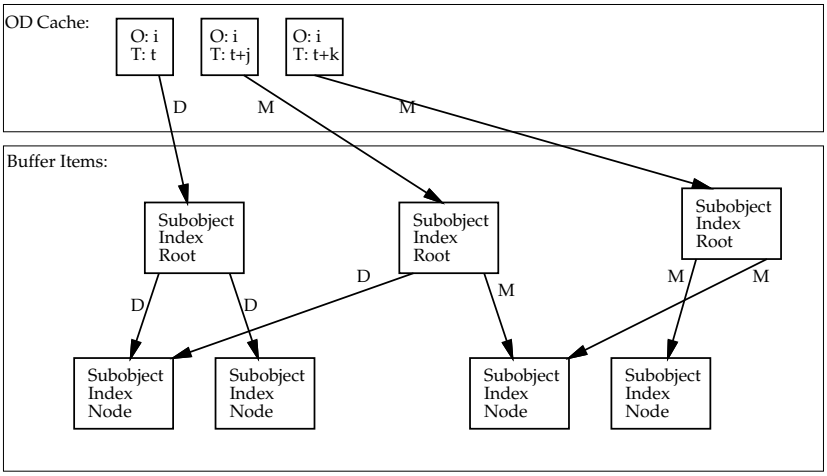


Figure 13.5: Large granularity buffer indexing. *O: i* denotes OID, and *T: t* denotes timestamp. The arrows are references, where D means a log address (in this case a disk address), and M means main-memory address.

Large Granularity Buffer Replacement

The large granularity buffers are similar to traditional disk page buffers, where an item is retrieved from disk to the buffer on demand. However, the writeback of dirty items is more complex, depending on the type of item:

- **PCache nodes:** The only references to PCache nodes are the pointers in main memory. There are no other dependencies, so they can be written back when necessary. However, if there are dirty ODs in the OD cache that are within the range covered by the PCache node that is to be replaced, they are inserted into the node if there are ODs that can be discarded from the PCache node.

- **TIDX index nodes:** Writing nodes from a tree-based index back to the log has to be done more carefully. Because of the log-only strategy, the log address of an index node changes after it has been written back to the log, as it will be written to a new location. When an index node has been written, the pointer in its parent node will still be pointing to the previous version. Therefore, the parent node has to be rewritten as well. This goes recursively to the root of the index tree, which means that the index has to be written bottom-up. To reduce the cost, dirty sibling nodes are written together, so that we avoid having to write their parent nodes several times.

  When writing TIDX leaf nodes, we first check the relevant resident PCache nodes if there are any dirty ODs that could be placed in the leaf node. In this way, the number of inserts into the tree at a later time can be reduced.

- **Subobject index nodes:** For subobject index nodes, we have the same consideration as for the TIDX index nodes regarding internode dependencies. In addition, the subobject index-tree nodes have to be written in a way that maintains the versioning consistency. This is easiest achieved by writing the oldest nodes first, effectively writing the subobject index nodes in commit timestamp order.

Number of live bytes
Segment state (CURRENT, DIRTY, CLEAN, SWAPPED, VDT, TEMP)

Table 13.6: Entry in the segment status table.

- **Subobjects:** Dirty subobjects can be written at any time before or during the transaction commit.

13.2.4 Segment Status Table Buffer

Information about the segments status is kept in main memory during normal operation, in the *segment status table* (SST). Normally, all of the segment status table will fit in this buffer. However, it is intended that the system's use of tertiary storage should be as transparent as possible. To achieve this goal, tertiary storage segments are treated similarly to secondary storage segments. With tertiary storage, the number of segments can be very high. In this case, it is possible to keep only part of the table in the buffer, and load parts of it on demand.

Table 13.6 summarizes the contents of an entry in the SST. A segment can be in the states *clean*, *current*, or *dirty*, as described in Section 5.1. Some segments are also used for other purposes than being part of the log. This is the case for segments storing the VDT (segment state VDT), and segments used to store intermediate results during query processing (segment state TEMP). We keep some statistics for each segment to help us decide which segments to clean: the number of live bytes, number of read accesses, and last access time. If new cleaning heuristics are developed (see Section 12.7.2), more information can be included in this table if necessary. Examples of such information is segment write time, number of read accesses, and the time of last access. More information in the SST can give more efficient cleaning, but the table has to be written once each checkpoint interval, so we also want to keep it as small as possible. As described here, using 512 KB segments, the size of the SST would be less than 10 KB for each GB of log. This is a reasonable size.

One thing that was considered, was to use logical instead of physical segments, as was done in the Spiralog file system [102, 212]. With logical segments, the log write address is monotonically increasing, and we would use a segment map to map between logical and physical segment. The advantage of logical segments is that segments can be migrated to other devices, for example tape, without having to update the metadata (in out case the OIDX). However, while logical segments can be very useful when all (or most) data in a segments has the same access characteristics, which can be the case in a file system, a segment in a DBMS is likely to include a larger degree of unrelated data. The mapping could also be quite costly. When using subsegments, you would need a mapping for every subsegment, not only each segment, and the map could grow very large.

13.3 Summary

We have in this chapter described the most important physical data structures in Vagabond. The main goal was to describe the most important aspects. In an implementation, more details are obviously needed. When implementing the main-memory data structures described in this chapter, attention has also to be paid to synchronization and data protection between threads and processes.

Part III

Analysis and Conclusions

Chapter 14

Analysis of the Log-Only Approach

In the previous chapters we have described the principles behind the log-only approach, and presented the design of the Vagabond log-only ODBMS. In this chapter, we compare the performance of a log-only based ODBMS (LO-ODBMS), with a traditional page-server ODBMS based on in-place updating (IPU-ODBMS). We give an introduction to analytical modeling, and present cost models for the IPU-ODBMS and LO-ODBMS approaches. The most important part of these models, the buffer models, are validated by comparing the models with simulation results. The IPU-ODBMS and LO-ODBMS models are used to compare the performance of the in-place update and log-only approaches, with different workloads and access patterns.

14.1 Analytical Modeling

Simulations and analytical modeling have been used extensively in database related research to compare different algorithmic and architectural options. In evaluations of complex systems, simulations have been most commonly used, mainly because of their proximity to implementation and the problem of incorporating concurrent events in analytical models. However, simulations have some very important shortcomings: 1) they are *time consuming*, and as a result only a small part of the parameter space can be explored, and 2) *it is not always easy to explain the results*! If analytical models are used, we can study the performance with many different combinations of parameters in a short time. It is also easier to determine dependencies and explain results with the help of analytical models. Analytical modeling can also be used together with simulations, to determine which parameters should be used, and to help explain the results of the simulations.

We have also deeply appreciated analytical modeling as a powerful tool in the design of Vagabond. Several bottlenecks were identified using the analytical models, and many of the solutions were also developed with its help.

Analytical modeling in database research has mostly focused on I/O costs. This is done under the assumption that I/O is the bottleneck. This is also our approach, but we have also incorporated buffer performance to a greater extent than before. This is necessary as a result of increasing amounts of main memory available for buffering.[1]

The aim of the analytical study is to compare the traditional in-place update approach with the log-only approach. Because of this, we focus on server performance only.

[1] It is very interesting (and alarming!) that many papers using the analytical approach *do not* include buffering aspects, or only do some very simplified assumption, not taking into account access skew. This is particularly evident in index related papers.

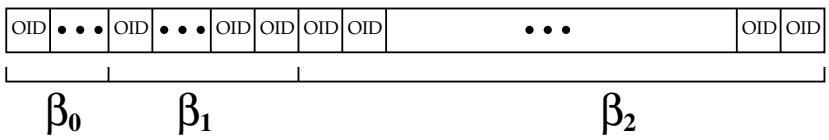


Figure 14.1: Logical access partitions.

14.2 Cost Model

Analytical modeling in database research has mostly focused on I/O costs. This has been the most significant cost factor, and the CPU processing has gone in parallel with disk transfer, making the CPU cost "invisible". With increasing amounts of main memory available, this is not necessarily correct. In that context, CPU cost and memory-to-memory transfer is important as well. Therefore, results from using the analytical model presented in this chapter when all data in the database fits in main memory should be taken with a large grain of salt, and we only provide results for memory sizes smaller than half the database size. However, even if the actual performance would not be as high as indicated by the model, the performance should still be much higher using the log-only approach, because we in this case can use the maximum disk write bandwidth.

The main purpose here is to study the disk bottleneck, and CPU cost should not affect the qualitative results in this thesis. We consider CPU costs to be orthogonal to disk costs. Another important reason for not including CPU costs in our models, is that this would be a much more complicated issue, and errors in the CPU cost models could easy hide important results. This means that while our model is an indicator of disk costs, CPU costs should be included in a model intended to be used to analyze the total performance of a system.

We use a traditional disk model, where the cost of reading a block from disk is the sum of the start up cost T_{start} and the transfer cost $T_{transfer}$. In our model, the average start up cost is fixed, and is set equivalent to t_r, the time it takes to do one disk revolution. The transfer cost is directly proportional to the block size, and is equivalent to reading disk tracks contiguously, i.e., transfer cost is equal to $\frac{b}{V_s}t_r$, where b is the block size to be transferred, and V_s is the amount of data on one track. For very large blocks, it is likely that several tracks and also tracks on different cylinders are read contiguously. This implies positioning, but we assume that the time used for this is insignificant compared to the transfer time. Sectors, typical 512 bytes, are the smallest addressable unit on a disk, and this implies that this is the smallest amount of data that can be read. Thus, the total time it takes to transfer a block of b bytes is:

$$T_B(b) = \begin{cases} t_r(1 + \frac{512}{V_s}) & \text{if } b \leq 512 \\ t_r(1 + \frac{b}{V_s}) & \text{if } b > 512 \end{cases} \tag{14.1}$$

To make the model less complex, we have not taken into account disk page overhead. We have also excluded some of the segment overheads, but we have included the ODs that have to be written together with the objects in the segments.

Set	β_0	β_1	β_2	α_0	α_1	α_2
2P8020	0.20	0.80	-	0.80	0.20	-
2P9505	0.05	0.95	-	0.95	0.05	-
3P1	0.01	0.19	0.80	0.64	0.16	0.20
3P2	0.001	0.049	0.95	0.80	0.19	0.01

Table 14.1: Partition sizes and partition access probabilities for the access patterns used in this chapter.

14.3 Object Access Model

We assume accesses to objects in the ODBMS to be random, but skewed (some objects are more often accessed than others). We assume it is possible to (logically) partition the range of OIDs into partitions, where each partition has a certain size and access probability. This is illustrated in Figure 14.1 (note that there is no correlation between OID and partition, i.e., OID=1 can be in partition 2, OID=2 can be in partition 1, etc.).

Our access model is the same as used by Bhide, Dan and Dias in their LRU buffer analysis [16], to be described in Section 14.4. This model, based on the independent reference model, considers a database with a size of N items (for example pages or objects). The database can be partitioned into p partitions. Each partition contains β_i of the items, as a fraction of the total database size or number of objects (see Figure 14.1), and α_i of the accesses are to each partition. The distributions *within* each of the partitions are assumed to be uniform, and all accesses are assumed to be independent. We denote an access pattern (partitioning set) for a set of items as Π. Π has p partitions, and the following has to be true:

$$\sum_{0}^{(p-1)} \alpha_i = 1.0$$

$$\sum_{0}^{(p-1)} \beta_i = 1.0$$

The studies in this chapter use four access patterns, summarized in Table 14.1. The first access pattern, Π=2P8020 is the traditional 80/20 access pattern, where 80% of the accesses go to 20% of the database. In the second access pattern, Π=2P9505, we have a smaller/hotter hot-spot partition. Even though the 80/20 model is applied in many analysis and simulations and is satisfactory in the analysis of many problems, it has a major shortcoming: when applied to calculate the number of distinct objects accessed, it gives a much higher number of distinct accessed objects than in a real system. The reason is that for most applications, inside the hot-spot partition (20% in this case), there is an even hotter and smaller partition, with a much higher access probability. This has to be reflected in the model, and is incorporated into the access patterns 3P1 and 3P2, which each consists of three partitions. In both access patterns, the 20% hot-spot partition from the 80/20 model is further partitioned. In 3P1 the 20% hot-spot partition is further divided into a 1% hot-spot partition and a 19% less hot partition. The last access pattern 3P2 resembles the access pattern close to what we expect it to be in future ODBMSs, with a large cold partition, consisting of old versions.

Some people might question the validity of the assumptions above, and wonder whether such a large amount of requests going to a small area is a realistic assumption. To show that this is quite

Partition Fraction	Partition Size (Distinct Words)			# of Words in the Book			Access Probability α_i		
	DB	EG	FS	DB	EG	FS	DB	EG	FS
$\beta_0 = 0.01$	241	25	94	432188	16341	87174	0.66	0.42	0.50
$\beta_1 = 0.19$	4583	465	1774	178910	17744	64640	0.27	0.46	0.37
$\beta_2 = 0.80$	19294	1960	7470	44830	4435	23531	0.07	0.12	0.13

Table 14.2: Partition sizes and partition access probabilities for three analyzed books, which shows that our assumptions regarding access pattern can be justified.

reasonable, even on non-temporal data, we will give an example. Imagine a spell checker, which spell checks a document by comparing each word in the document against the words in a dictionary which is accessed by an index. Each word in the dictionary can be though of as an object, while each word in the document that is spell checked can be considered an object access (we have to look up this word in the dictionary). The access probability for a book with W words in total, and N distinct words, can be found as follows:

1. Define partition sizes β_i, for example 0.01, 0.19 and 0.80 as in the 3P1 access pattern.
2. Count the number of occurrences for each distinct word.[2]
3. Sort the word/occurrence counts tuples on occurrence counts. The $\beta_0 N$ most frequent words are in partition 0, the next $\beta_1 N$ words in partition 1, and the $\beta_2 N$ less frequently used words in partition 2.
4. The access probability for partition i is the sum of the occurrence count for each word in the actual partition, divided on W.

As an example, we have used three books: a Danish bible (DB), an English version of the Genesis (EG), and a Norwegian file systems book (FS). If we consider the number of distinct words in each of the books as dictionaries (24118, 2450 and 9338 unique words, respectively), and do a spell check of the books (655928, 38520, and 175345 word accesses), we obtain the access probabilities for the 1%/19%/80% partitions as summarized in Table 14.2. They access probabilities show that our assumptions regarding access probabilities can be justified, and is also consistent with a comparison with the Zipf distribution.

14.4 The BDD LRU Buffer Model

In our analysis, we need to estimate the buffer hit probability in an LRU managed buffer given a certain access pattern. We also need to estimate the number of distinct items accessed given a certain number of accesses. Both can be computed with equations from the LRU buffer model developed by Bhide, Dan and Dias, hereafter called the BDD model. In this section we present the main results and equations from the model. The derivation and details behind the equations can be found in [16].

[2]Note that each conjugation of a word will be present in the spell checker dictionary. The actual number of words as would appear in a traditional dictionary would be smaller.

Distinct Items. After n accesses with access pattern Π to a database containing N items, the number of *distinct* items that has been accessed is (p is the number of partitions as defined in Section 14.3):

$$N_{distinct}(n, N, \Pi) = \sum_{i=1}^{p} N^i_{distinct}(n, N, \Pi)$$

where $N^i_{distinct}(n, N, \Pi)$ is the number of distinct items from partition i that have been accessed:

$$N^i_{distinct}(n, N, \Pi) = \beta_i N(1 - (1 - \frac{1}{\beta_i N})^{\alpha_i n}) \tag{14.2}$$

Buffer Hit Probability. When the number of accesses n is such that the number of distinct data items accessed is less than the buffer size B (the number of items that fits in the buffer), $\sum_{i=1}^{p} N^i_{distinct} \leq B$, the buffer hit probability for partition i is:

$$P_i(n, \Pi) = 1 - (1 - \frac{1}{\beta_i N})^{\alpha_i n}$$

and the overall buffer hit probability is:

$$P(n, \Pi) = \sum_{i=1}^{p} \alpha_i P_i(n, \Pi)$$

The steady state average buffer hit probability can be approximated to the buffer hit ratio when the buffer becomes full, i.e., n is chosen as the largest n that satisfies :[3]

$$\sum_{i=1}^{p} N^i_{distinct}(n, N, \Pi) = B$$

We denote the average buffer hit probability as:

$$P_{buf}(B, N, \Pi) = P(n, \Pi) \tag{14.3}$$

where n is chosen as described above.

Buffer Hit Probability in a Buffer with Locking. The BDD model assumes that all items in the buffer are eligible for replacement. However, this is not the case in a buffer with dirty items, for example in a page buffer, object buffer, or OD cache. To avoid a very costly synchronous writeback of items in the buffer, items are written back to the database/index asynchronously in the background, using some strategy to reduce disk arm movement. This means that some of the items in the buffer are locked and cannot be discarded until they have been written back to disk.

The fraction of the items in the buffer that are locked depends on the writeback rate. A high writeback rate results in less locked items and increases the buffer hit rate, but in general it also increases the average writeback cost. We denote the fraction of the items in the buffer that are allowed to be dirty as F_{dirty} (which means that the writeback rate should be high enough to keep the number of dirty items less than $F_{dirty}B$). To model the behavior of the dirty items in the buffer we use a simple extension of the BDD mode, which we call the *DCOMP model*. In the DCOMP model we use

[3]Binary search can be used to find n, because $N^i_{distinct}$ are monotonically increasing functions of n.

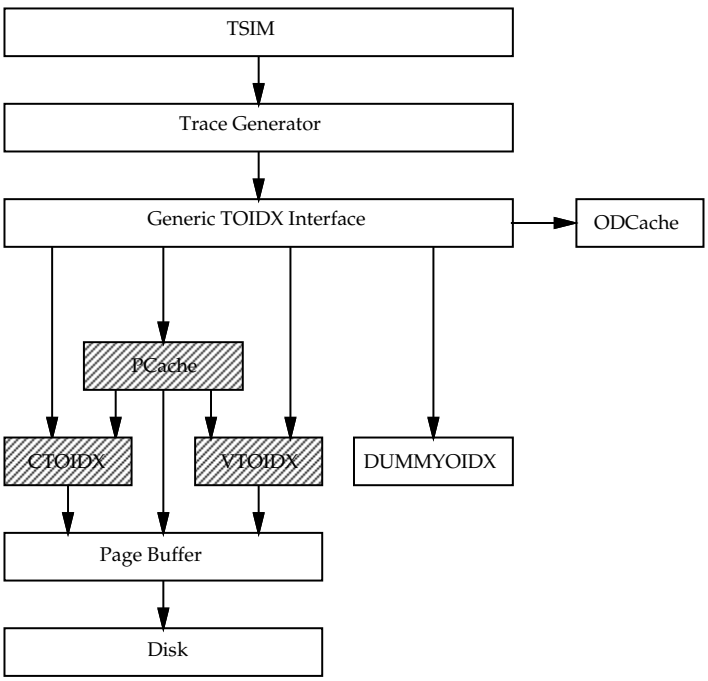


Figure 14.2: TSIM architecture.

Equation 14.3, but exclude the slots in the buffer used by the dirty items when we calculate the buffer hit ratio:

$$P_{bufD}(B, N, \Pi, F_{dirty}) = P_{buf}((1 - F_{dirty})B, N, \Pi) \tag{14.4}$$

This equation is valid with relatively low values of F_{dirty}. In Section 14.4.7 we will study how larger values of F_{dirty} affect the accuracy of the model.

14.4.1 The Validity of the BDD Model in Our Context

The assumptions behind the BDD model include a strict LRU policy, only read accesses, and independent accesses. However, in our model, we assume a low overhead clock algorithm used as an LRU approximation, and a mixed workload with both read and write accesses. Although it is known that for sufficiently large buffers the clock algorithm is a good approximation of the LRU strategy [61], the introduction of locked items in the buffer makes it necessary to do a new study of the accuracy of the modified model.

In order to validate the analytical model, we have done an implementation of the OD cache and a simulated temporal OIDX (TOIDX). We will now describe the simulator which we used in the validation study, and then describe the results from our simulations.

14.4.2 An Overview of the Simulator

The TOIDX simulator (TSIM) is a toolbox for studies of different TOIDXs and additional structures like the PCache. Figure 14.2 gives an overview of the architecture of TSIM. All the modules neces-

sary to study the OD cache are implemented, but the indexes are not yet implemented (illustrated by hatched boxes in the figure). We will now briefly describe the main modules in TSIM.

Trace Generator. The trace generator generates object access requests (OID/TIME requests), which are handled by simulator. Requests from the trace generator can be generated on the fly, based on defined access patterns, or they can come from traces from real programs running against a temporal ODBMS. The results reported in this thesis is from an artificially generated workload, which will be described in more detail in Section 14.4.3.

Generic TOIDX Interface. This is a generic interface to the index. All TOIDX implementations and the PCache share the same interface, so that it is easy to decide at run time which one to use. Control and use of the OD Cache (insert, lookup, flush, etc) is also done from this module.

OD Cache. This is an implementation of the OD cache described in Section 13.2.1, with cache replacement according to the clock algorithm.

PCache, CTOIDX and VTOIDX. These modules, when implemented, will contain implementations of the PCache and TOIDXs, including a composite TOIDX (CTOIDX) and the VTOIDX, described in Section 8.3.2 and Section 8.4.

In the current version of the simulator, a "dummy OIDX" (DUMMYOIDX) is used. The DUMMYOIX has the same interface as the other TOIDXs, but does not actually store the contents on disk pages, they are only kept in a main-memory structure.

Page Buffer. This module implements the main-memory buffer used for disk pages. This module makes it easy to see how different buffer sizes affect performance, and get buffer performance statistics. Pages in the buffer are kept in a linked list, and managed according to the LRU algorithm.

Disk. This is the disk subsystem. All accesses to the disk module are done through the page-buffer module. The disk module can operate in one of two modes:

1. Simulated disk accesses. In this case, main memory is used to store the disk pages. This is useful if we only want to count disk accesses, or use an external analytical disk model.

2. Real disk accesses. In this case, pages are actually read from and written to disk. This is useful when we want to study the actual performance.

14.4.3 Simulation Results

The workload consists of read, write, and create requests. There are no separate delete requests, but delete can in this context be considered a write operation, because a new OD will be generated (in the case of temporal data), or the OD removed (in the case of non-temporal data). The following parameters are used to specify the workload:

- Π: The access pattern (see Section 14.3). Both read and write accesses are done according to this access pattern.

- P_{write}: The probability that an operation is a write or create operation.

- $P_{write_temporal}$: The probability that a write operation is to a temporal object. The distinction between temporal and non-temporal object is important, because when a temporal object is updated, we create a new OD for the new version, but when we update a non-temporal object, we modify the existing OD.
- P_{new}: The probability that a non-temporal write operation creates a new object.
- N_{ver}: The average number of versions for each temporal object, i.e., number of versions with the same OID, but with different timestamps.
- F_{dirty}: The fraction of the ODs in the OD cache that are allowed to be dirty, i.e., not yet inserted into the TOIDX.
- N_{objver}: The total number of object versions in the database.

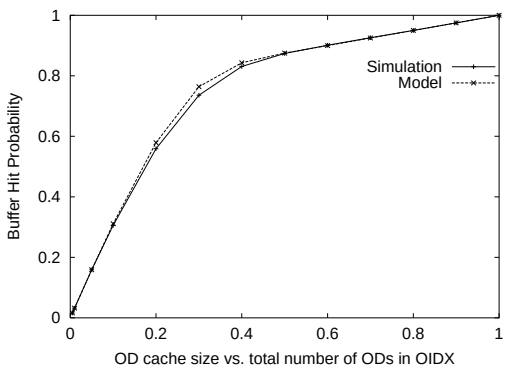
We have done the simulations with the access patterns in Table 14.1. Default values for the other parameters are $P_{write} = 0.2$, $P_{write_temporal} = 0.8$, $P_{new} = 0.2$, $N_{ver} = 5$, and $F_{dirty} = 0.2$. When these parameters are used in a simulation, the result is a database where 5% of the object versions are non-temporal objects, 20% of the object versions are current versions of temporal objects, and the others are historical versions of the temporal objects. The total number of object versions has been set to $N_{objver} = 100000$. This might seem too small, but the difference between using $N_{objver} = 100000$ and a larger value of N_{objver} is only marginal.

The simulations have been run in two modes, *READ_DB* and *FILL_DB*:

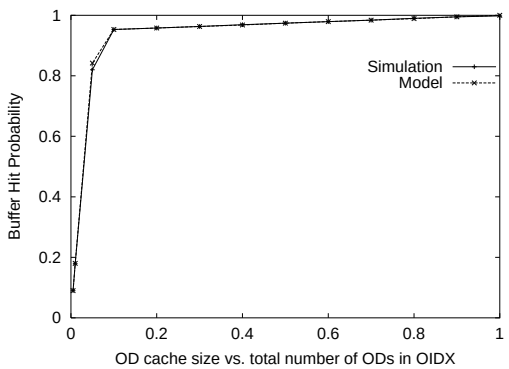
- In the *READ_DB* mode we are only interested in the hit rate for a read-only workload when the database is in a stable condition. When using this mode, we first insert all N_{objver} object versions into the database. After the object versions have been inserted, a large number of read accesses to the database are done to "warm up" the OD cache. After the warm-up phase, we assume the OD cache is in a stable condition, and we measure the hit rate from a number of read accesses.
- In the *FILL_DB* mode, we want to measure the hit rate under a more typical workload, with both read and write accesses applied to the system. One problem when doing this, is that the analytical models are based on a database in a stable condition. This is difficult to simulate in the case of a temporal database, because the number of object versions is continuously increasing. Although this problem could be reduced by including vacuuming into the simulation, this would make the access probability analysis more difficult. Instead, we do accesses to the database according to the parameters above, and only start measuring the hit rate when the database size is larger than $0.9 * N_{objver}$. As we shall see later, this gives acceptable results, except in the case where P_{write} is large.

14.4.4 Model Validation

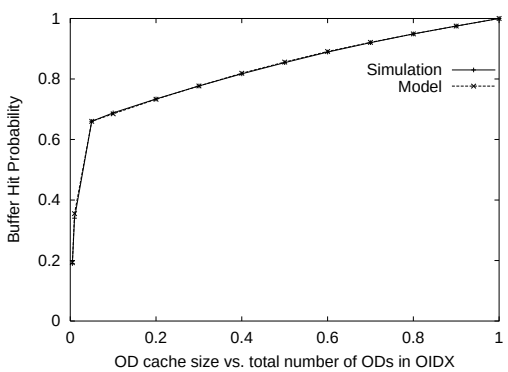
Figure 14.3 shows the simulated OD cache hit rate for read only accesses (obtained using the *READ_DB* mode), compared with the calculated hit rates based on the BDD buffer model. For all access patterns, the deviations between the model and the simulations are in general less than 1%. The only exceptions are for very small sizes of the OD cache compared to the total number of ODs in the database, and for a small range of OD cache sizes using the 2P8020 access pattern. The accuracy of the model in the case of a read only workload is as expected, because the accesses are done following the assumptions



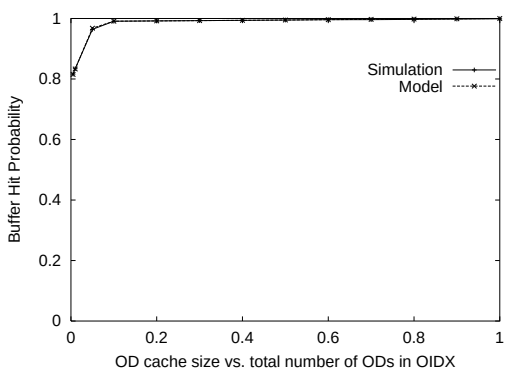
(a) 2P8020 access pattern.



(b) 2P9505 access pattern.

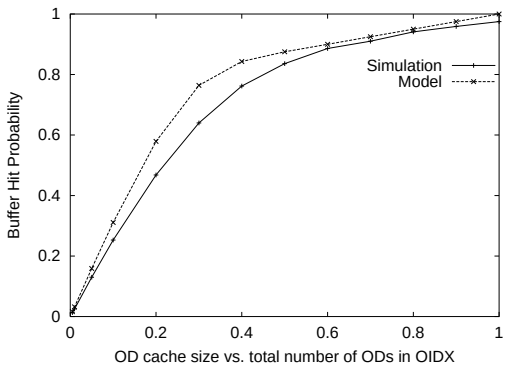


(c) 3P1 access pattern.

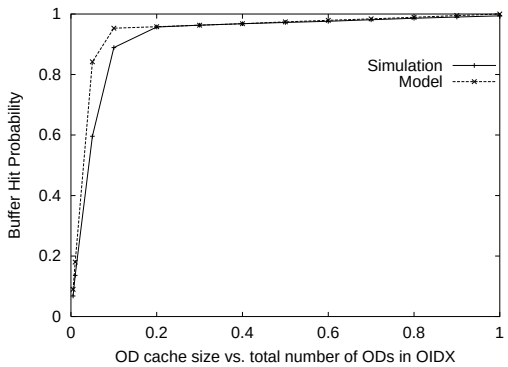


(d) 3P2 access pattern.

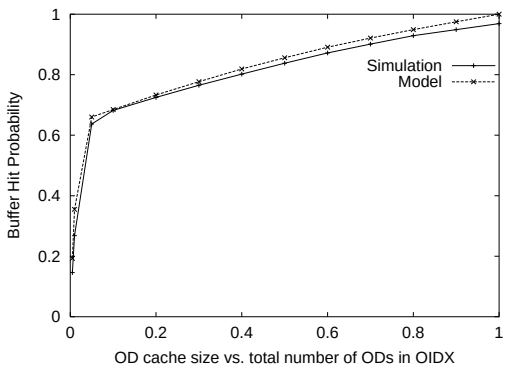
Figure 14.3: OD cache hit rate for read only accesses.



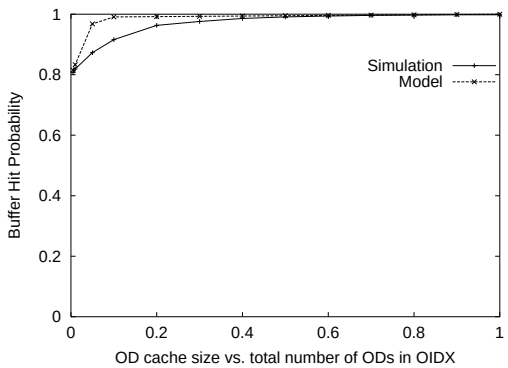
(a) 2P8020 access pattern.



(b) 2P9505 access pattern.



(c) 3P1 access pattern.



(d) 3P2 access pattern.

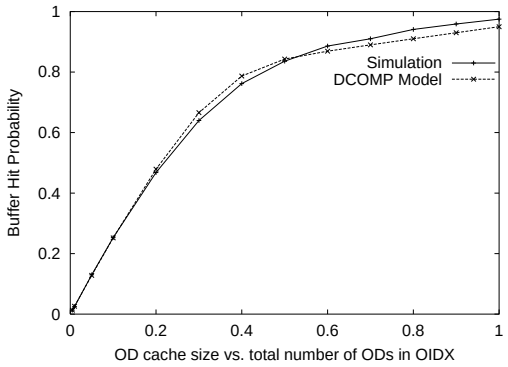
Figure 14.4: OD cache hit rate with mixed workload.

behind the BDD model. The reason for the few inaccuracies that can be observed, is that the OD cache uses the clock algorithm, which is only an approximation to the LRU algorithm.

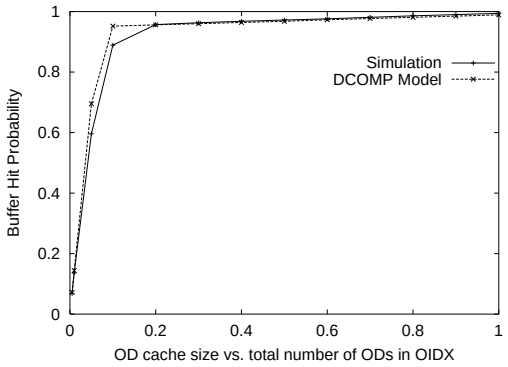
Figure 14.4 shows the simulated OD cache hit rate in the case of a workload of both read and write accesses (obtained from using the *FILL_DB* mode). In this case, the deviation between model and the simulation is much higher. The reason for this, is that dirty ODs are locked in the OD cache until they have been written to the TOIDX. Many of these dirty ODs are not hot-spot ODs, and would otherwise have been replaced in the OD cache.

To model the behavior of the dirty ODs, we exclude the slots in the OD cache used by the dirty ODs when we calculate the OD cache hit ratio, i.e., we calculate the hit ratio for a buffer less than the one we actually have. We name this modified model *DCOMP*. Figure 14.5 illustrates the much higher accuracy we obtain when using the DCOMP model.

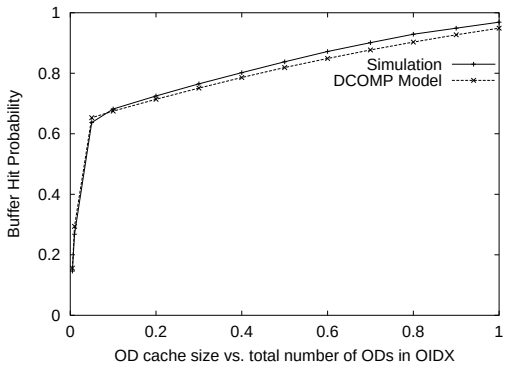
In order to be able to study in more detail the deviation between the model and the simulations, we will study the deviation between model and simulation, instead of the actual hit rates. The deviation



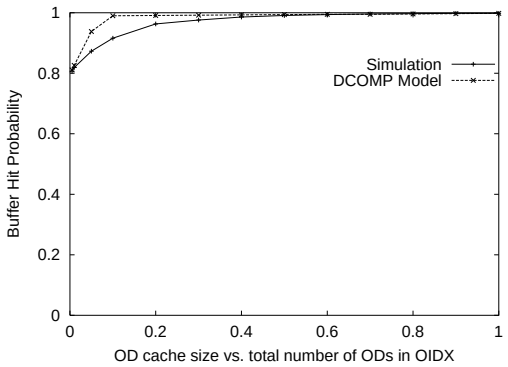
(a) 2P8020 access pattern.



(b) 2P9505 access pattern.

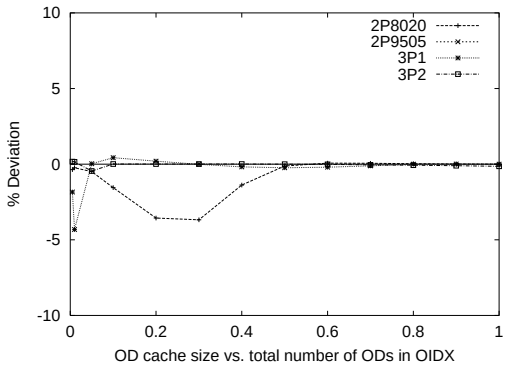


(c) 3P1 access pattern.

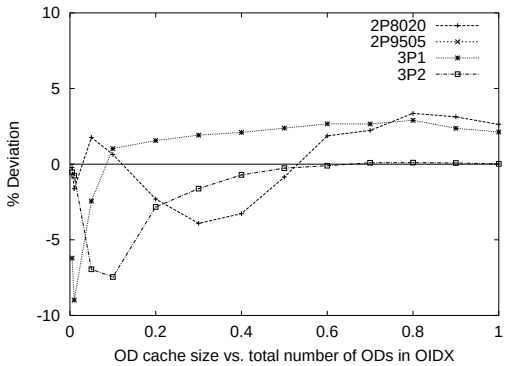


(d) 3P2 access pattern.

Figure 14.5: OD cache hit rate with mixed workload.

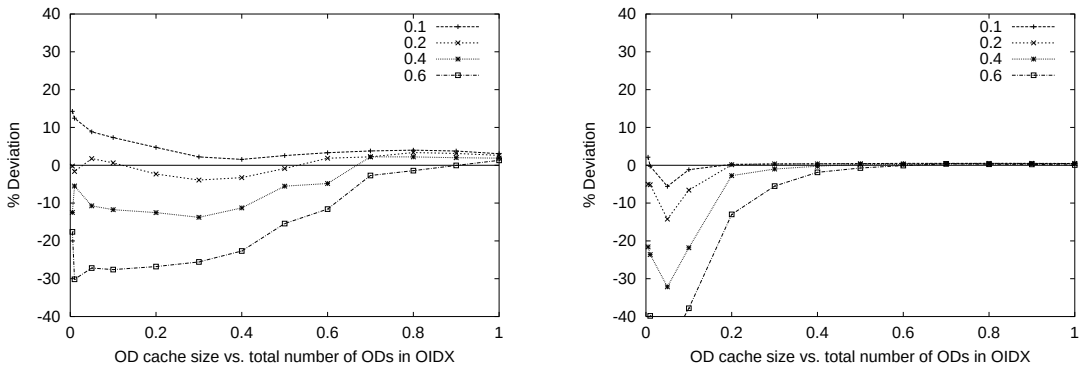


(a) Read only (*READ_DB*).



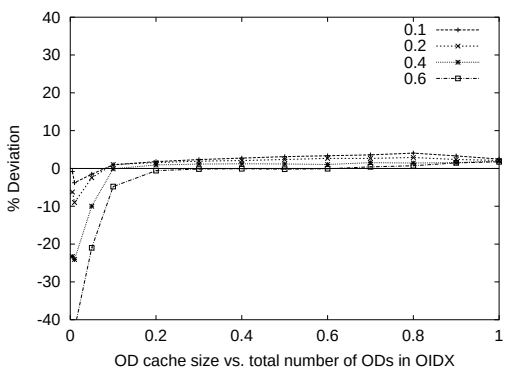
(b) Mixed workload (*FILL_DB*).

Figure 14.6: Deviation between simulation and the DCOMP model, using the default parameters.

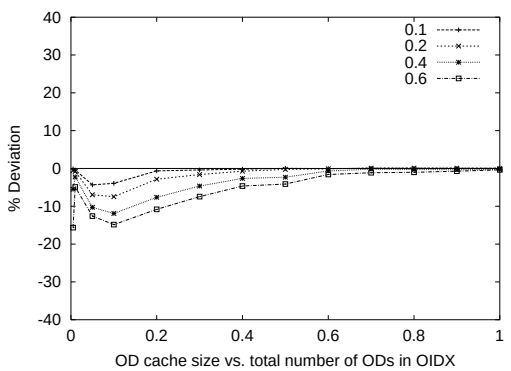


(a) 2P8020 access pattern.

(b) 2P9505 access pattern.



(c) 3P1 access pattern.



(d) 3P2 access pattern.

Figure 14.7: Deviation between simulations and the DCOMP model with different write rates.

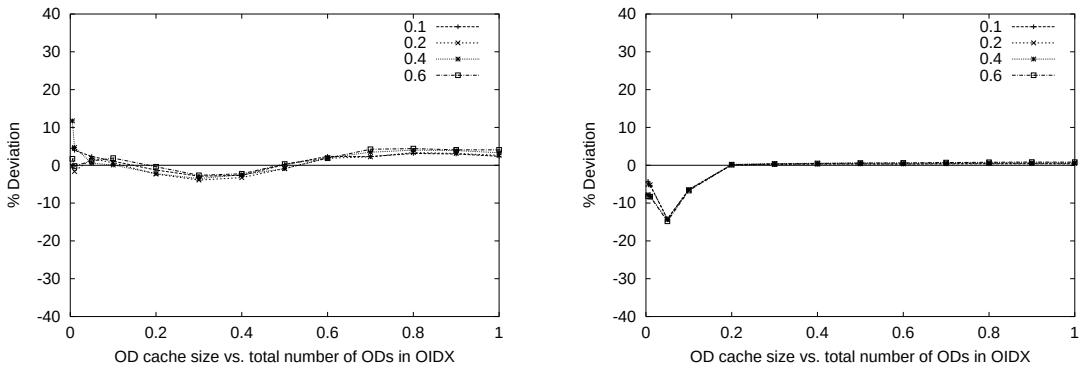
is calculated according to:

$$D = \frac{P_{buf}^{Simulated} - P_{buf}^{Model}}{P_{buf}^{Model}} 100\%$$

Figure 14.6 shows the deviation using the default parameters when simulating in the *READ_DB* and *FILL_DB* modes.

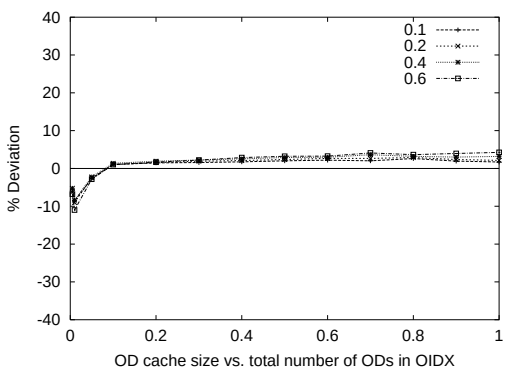
14.4.5 The Effect of Different Write Rates

As mentioned, the buffer models assume a stable condition. A high write rate (high P_{write}) breaks this assumption, as new objects (and ODs) are inserted into the database so fast that the OD cache does not necessarily stabilize. However, as long as the number of dirty ODs in the OD cache is low (set by the parameter F_{dirty}) and the DCOMP model is used, the high insert rate in itself is not the main contributor to the deviation between the model and the simulations. The deviations are a side effect of the way the simulations are done. The proportional size of the access pattern partitions are constant, which means that the number of ODs in the hot-spot partition is constantly increasing. In a

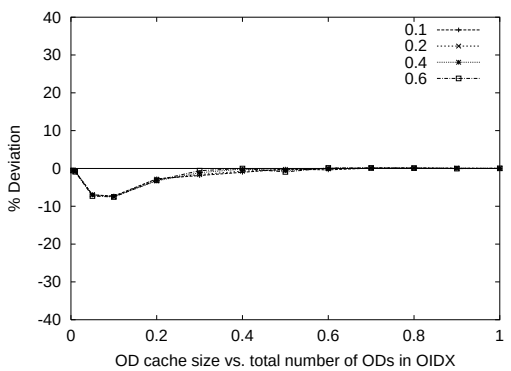


(a) 2P8020 access pattern.

(b) 2P9505 access pattern.



(c) 3P1 access pattern.



(d) 3P2 access pattern.

Figure 14.8: Deviation between simulations and the DCOMP model with different object create rates.

real system, we expect the number of ODs in this partition to be more constant. The deviations are illustrated by Figure 14.7, and are most evident for small sizes of the OD cache.

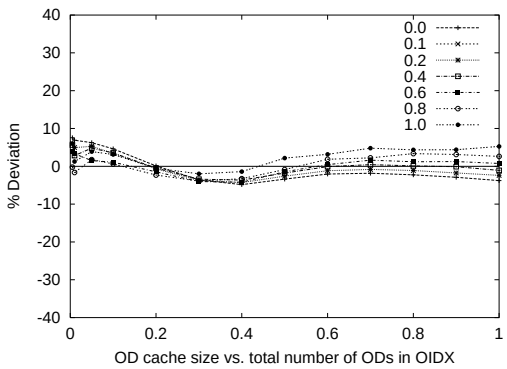
Although we have a significant deviation between simulations and model when using high write rates, we are not convinced that the buffer model really gives a false impression of the hit rate. Rather, the deviation here illustrate the problem of getting a "snapshot" of a stable database during simulations with a high insertion rate.

14.4.6 The Effect of Different Object Create and Temporal Data Access Rates

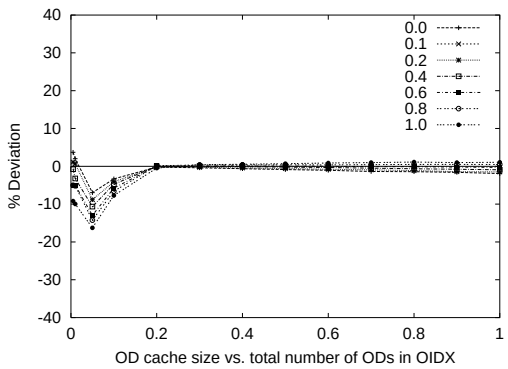
Figures 14.8 and 14.9 illustrate the effect different object creation rates and different proportions of accesses to temporal data have on the deviation. In both cases, the deviation is only marginal.

14.4.7 The Effect of Different Amounts of Dirty ODs in the OD Cache

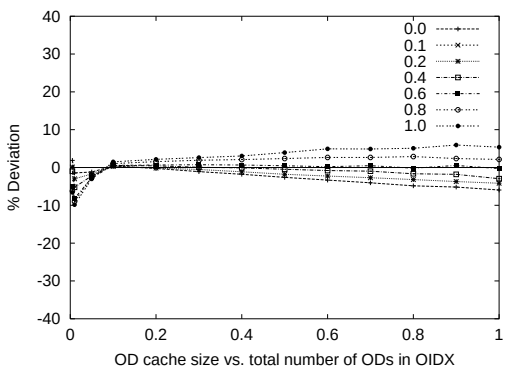
We assumed previously that the dirty ODs in the OD cache did not contribute to the hit rate, because many of these ODs would be "cold" ODs that would have been discarded if they were not dirty. If we



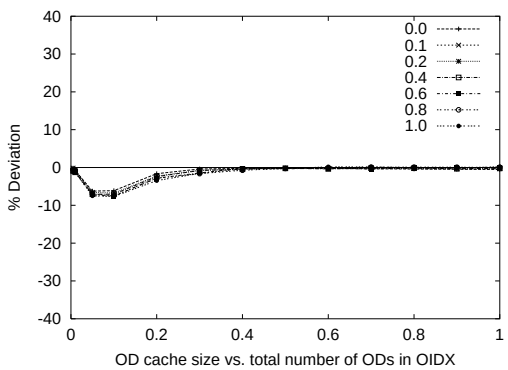
(a) 2P8020 access pattern.



(b) 2P9505 access pattern.

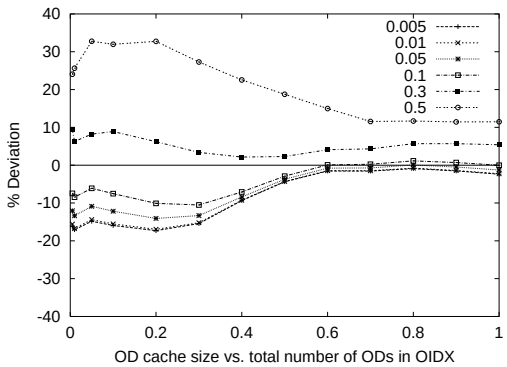


(c) 3P1 access pattern.

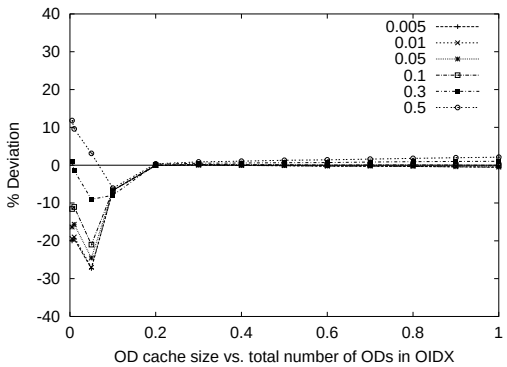


(d) 3P2 access pattern.

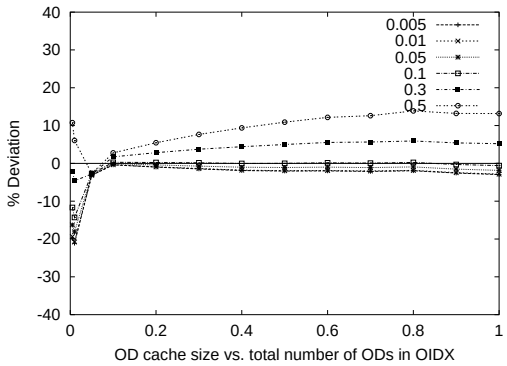
Figure 14.9: Deviation between simulations and the DCOMP model with different amounts of temporal data.



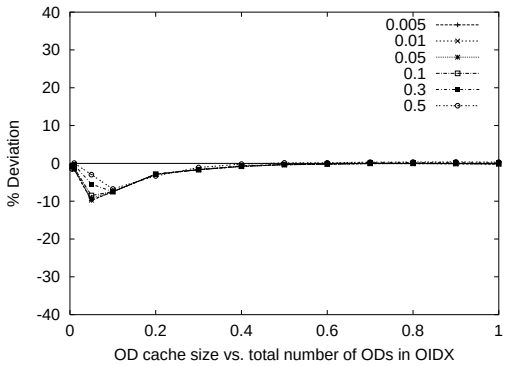
(a) 2P8020 access pattern.



(b) 2P9505 access pattern.



(c) 3P1 access pattern.



(d) 3P2 access pattern.

Figure 14.10: Deviation between simulations and the DCOMP model with different amounts of dirty data in the OD cache.

allow a large percentage of the ODs in the OD cache to be dirty, this will result in an underestimation of the hit rate, because a larger amount of these dirty ODs will actually be read and contribute to the simulated hit rate. This is illustrated on Figure 14.10. A fraction of dirty ODs larger than approximately $F_{dirty} = 0.3$ results in an unacceptable high deviation between the simulations and the model. However, in practice F_{dirty} will be less than 0.3 because a large number of dirty ODs (or objects) means a large checkpoint interval (to keep recovery time short, the checkpoint interval should not be too long). In practice, F_{dirty} will also be much less than the default value used in this study, resulting in less deviation.

14.4.8 Discussion

We have now compared the fine granularity buffer model with simulations using different database and workload parameters. We have shown that in most cases, the deviations between the model and the simulations are acceptable.

The clock algorithm is only a good approximation for LRU if the number of items that fits in a cache is sufficiently high [61]. This is also evident for our simulations: with a small OD cache size, the comparison shows high deviation. The other situation where we get a very high deviation, is when we allow a large number of dirty ODs in the OD cache (high value of F_{dirty}). In this case, the DCOMP model over-compensates for the dirty ODs. Whether the deviations for the other parameter values are acceptable, depends on how much accuracy is expected.

14.5 Assumptions Behind the ODBMS Models

In this section we outline the assumptions and prerequisites for the analytical models of IPU- and LO-ODBMSs that will be presented in the following sections. To make the description of the models easier to read, we have summarized the system parameters and functions in Table 14.3. Parameters and functions specific for only one of the models are summarized in separate tables (Tables 14.4 and 14.5).

14.5.1 General Assumptions

In our analysis, we limit the discussion to storage and retrieval of the objects only, under the assumptions that:

1. We have enough disk available to avoid the OIDX operations becoming a bottleneck.
2. In a traditional IPU-ODBMS, the OIDX will be based on in-place updating. In an LO-ODBMS, the OIDX can either be stored log-only, or it can be stored separately, and updated in-place. By not including the OIDX in the analysis, we assume that OIDX operations in an LO-ODBMS will have the same costs as in an IPU-ODBMS.
3. The database in a stable condition, with N_{obj} object versions.

To be able to analyze and compare the log-only approach with a traditional page server employing in-place updating, we need a model for each of the approaches. The models will include object storage and retrieval costs. The average time to read an object is denoted $T_{readobj}$, and the average time to write an object is denoted $T_{writeobj}$. It will be understood from the context whether the cost function applies to an IPU- or LO-ODBMS. We restrict this analysis to objects smaller than one disk page.

Parameter/ Function	Definition	Default Value/ Equation
t_r	Disk revolution time	$\frac{1}{167}s$
C	Clustering factor	(14.5)
M_{obuf}	Size of object buffer/object page buffer	-
$N_{distinct}(n, N)$	Number of distinct object/pages	(14.4)
N_{obj}	Number of objects in the database	S_{DB}/S_{obj}
$P_{buf}(M, N)$	Buffer hit probability	(14.3)
P_{bufD}	Buffer hit probability in a buffer with locked items	(14.4)
P_{new}	Probability that a write creates a new object	0.2
P_{write}	Object write probability	0.2
$P_{write_temporal}$	Probability that a write operation is to a temporal object	0.8
P_{RC}	Probability that a read operation is for the current version	0.9
R_{comp}	Compression ratio	1:1 (No compression)
S_{obj}	Average size of an object	208
S_{od}	Size of an OD	36 B
$S_{overhead}$	Overhead for an item in the buffer	12 B
S_{DB}	Database size	8 GB
$Speedup$	Speedup	(14.15)
T_{access}	Average object access cost	(14.14)
$T_{readobj}$	Average time to read an object	(14.8, IPU) and (14.11, LO)
$T_{writeobj}$	Average time to write an object	(14.9, IPU) and (14.13, LO)
T_B	Cost (time) of transfer of one block	(14.1)
T_S	Cost of writing b bytes sequentially	$\frac{b}{V_s}t_r$
V_s	Disk track size	120 KB

Table 14.3: Summary of system parameters and functions used in the models.

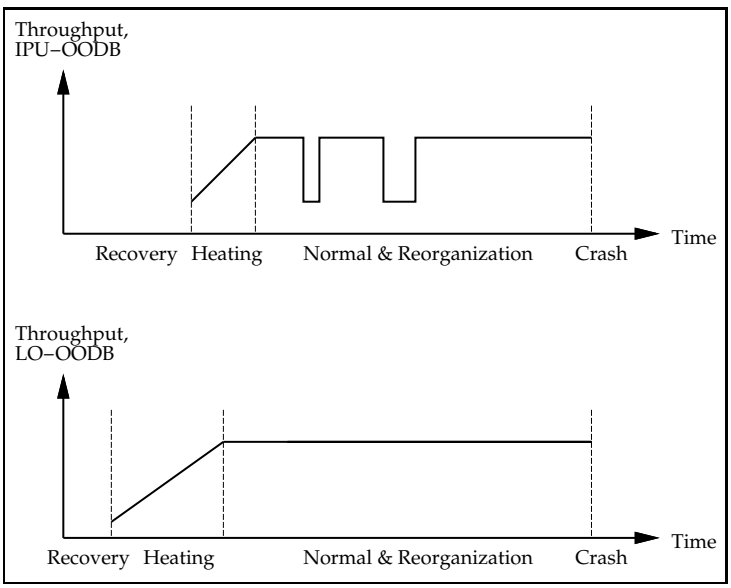


Figure 14.11: Throughput during different operational phases in an IPU-ODBMS and an LO-ODBMS.

In both models, we assume the total amount of available buffer memory to be M. In the case of the IPU-ODBMS, this memory is used for buffering object pages, OIDX pages, and object descriptors (OD cache). In the case of the LO-ODBMS, this memory is used for buffering objects, OIDX pages, and object descriptors (OD cache). Because in this analysis we do not consider index performance, we will only consider the memory available for objects and object pages. We denote this as M_{obuf}.

Objects and the OIDX can be partitioned over several disks, and in both systems modeled here, the performance can be increased by adding more disks. To simplify the analysis, we assume that only two disks are available for object storage. This is the smallest number of disks needed to be able to recover from media failures. In the IPU-ODBMS, one of the disks is used for data, and the other one is used for the log. In the LO-ODBMS, we use the two disks in a RAID 1 configuration (mirroring). This means that we get the write performance of one disk, but the read performance of two disks.

Some optimizations for the LO-ODBMS described previously in this thesis are not included in the model, for example signatures and delta objects. Using these optimizations could improve the relative performance. For example, storing signatures in the OD and using them to reduce query cost, would be especially beneficial for the log-only approach. The reason for this, is that when using signatures to filter out objects, we often loose the benefits of the object clustering in an IPU-ODBMS approach. The clustering is often the key to performance in page servers, and loosing the benefits from the clustering significantly reduces the gain from using signatures. In an LO-ODBMS, higher gain from using signatures can be achieved, because clustering is not so much of an issue, and hence, there is less to loose.

14.5.2 Operational Phases

It is important to note the phases a DBMS goes through. The performance *and the duration* of each of these phases is important, illustrated in Figure 14.11. The sum of the duration of these phases is the time between DBMS crashes.

Function	Definition	Equation/ Default Value
C	Data clustering factor, $1/N_{o_page} \leq C \leq 1.0$	0.20
N_{CP}	Checkpoint-interval length[4]	2 MB$/S_{obj}$
N_{o_page}	The average number of objects on each page	S_P/S_{obj}
P_{buf_opage}	Probability of finding a page in the buffer	(14.6)
S_P	Page size in page server[5]	4 KB
T_P	Cost of random read/write a page	$T_B(S_P)$
$T_{readpage}$	Average cost of reading a page	(14.7)

Table 14.4: IPU-ODBMS specific parameters and functions.

1. **Recovery:** In this phase the system is made consistent after a crash.

2. **Heating:** When the system is made available, a lot of buffer misses will be experienced. In this phase, and while we "warm up" the buffer, the performance can be relatively low. Some buffer heating might also have been done in the previous phase if recovery was needed.

3. **Normal:** The buffer now contains the hot set, and with sufficient buffer capacity, the throughput in this phase is ultimately limited by the write capacity. This clearly favors a system with a mostly sequential write pattern. Using in-place updating, even when all data fits in the buffer, random writes will be necessary when installing data into the database. However, some kind of reorganization has to be done periodically, or concurrently with normal operation. With a traditional approach, the performance of this phase is also dependent of how well clustered the data is, as buffering usually is done pagewise. With an object buffer, this is no problem. We do not waste memory with unwanted data, that by coincidence happens to be on the same page as some more active data.

4. **Reorganization:** With changing access patterns and schema changes (for example objects that have increased in size), even a traditional system will become less clustered. In this case, a reorganization is needed. This users of the system will at this time experience a sharp drop of throughput. Not all systems support on-line reorganization, these systems will be unavailable during the reorganization. In a log-only system, reorganization can easily be done concurrently with normal work, objects to be moved are simply placed in the output stream and written sequentially as the objects written by the ongoing transactions.

We will in this study concentrate on the normal operation phase, and only briefly discuss the performance doing the other phases.

14.6 Analytical Modeling of an IPU-ODBMS

[5]We keep the data volume written during a checkpoint-interval length, so that the actual number N_{CP} of objects written depends on the average object size. With the default object size $S_{obj} = 208$, $N_{CP} \approx 10000$.

[5]It is possible to use a larger page size than 4 KB a is used as default in our analysis, but that will increase the level of false sharing on the pages, and reduce the data clustering factor.

In an IPU-ODBMS, space is allocated for an object when it is created, and further updates to the object are done in-place. This implies that after an object update, the previous version of the object is not available unless it has been stored elsewhere.

In a temporal IPU-ODBMS based on traditional techniques, it is usually assumed that most accesses will be to the current versions of the objects in the database. To keep these accesses as efficient as possible, and benefit from object clustering, the database is partitioned. The current versions of objects are in one partition, in the *current database*, and the historical versions in another partition, in the *historical database*. When an object is updated in a temporal IPU-ODBMS, the previous version is first moved to the historical database, before the new version is stored in-place in the current database.

We assume that clustering is not maintained for historical data, so that all object versions that are moved because they are replaced by a new current version, can be written sequentially. This reduces the update costs considerably.

Not all of the data in a temporal ODBMS is temporal. For some objects, we are only interested in the current version. To improve efficiency, the system can be made aware of this. In this way, some of the data can be defined as non-temporal. Old versions of these are not kept, and objects can be updated in-place as in a non-temporal (one-version) ODBMS. For these objects, the costly OIDX update is not needed when the object is modified.

In a traditional page-server system, a dedicated disk is usually used for the log. In this way, disk seek is avoided when writing to the log. and it is also necessary in order to handle media failures. We assume the log writing costs are less than the other costs involved, so that when a separate log disk is used, we do not have to consider the cost of log operations in this analysis.

We have in this analysis assumed a space utilization of 100% on the object pages. This is a very optimistic assumption done on behalf of the IPU-ODBMS approach. In practice, the space utilization will be lower, as it is unwise to fill pages prematurely. Free space in pages makes it possible to store new objects that should be clustered together with old ones in a page, and makes it possible to maintain good clustering even when objects in a page get larger.

14.6.1 Clustering

In general, more than one object is stored on each disk page. To reduce the object retrieval cost, objects are often placed on disk pages in a way that makes it likely that more than one of the objects on a page that is read, will be needed in the near future. This is called clustering (see Section 3.3).

When requested data resides on pages, as is the case for page servers, or index entries resides in index nodes, and the granule requested is only a fraction of the item read or written (page or node), we can have a certain degree of clustering. We define the *clustering factor* C as the fraction of a page that is referenced during one transaction. If there are N_{o_page} objects on each page, and n of them will be used, the clustering factor when $N_{o_page} >= 1$ is:

$$C = \frac{n}{N_{o_page}} \tag{14.5}$$

It should be noted that clustering can be viewed from two sides, writing and reading. A clustering optimized for reading, will commonly be different from one optimized for updating. To avoid a too complex model, we will in the rest of this thesis assume read and write clustering to be equal, and clustering of objects on pages in a page server to be equal to index page clustering, denoting both as C.

14.6.2 Checkpoint Interval

Updating can be done in-place, with WAL. In that case, a transaction can commit after its log records have been written to disk. Modified pages are not written back immediately, this is done lazily in the background as a part of the buffer replacement and checkpointing. Thus, a page may be modified several times before it is written back. The update costs will be dependent of the checkpoint interval, defined as the number of objects that can be written between two checkpoints. This number of written objects, N_{CP}, includes *created* as well as *updated* objects. We assume that the buffer is large enough to hold pages written to several times during one checkpoint interval, i.e., we always use a value of N_{CP} where this is true. This is already true in many configurations, and will certainly be valid for future system with large amounts of main memory.

14.6.3 Object Read Cost

Even with several page requests at a time, and employing an elevator algorithm, the seek time will be significant. The cost of reading or writing a page of size S_P to/from the disk is equal to $T_P = T_B(S_P)$. When reading a page, the page may already be resident in the page buffer.[6] The probability for this is P_{buf_opage}:

$$P_{buf_opage} = P_{bufD}(\frac{M_{obuf}}{S_P + S_{overhead}}, \frac{S_{DB}}{S_P}, \Pi, \frac{N_P/2}{S_{DB}/S_P}) \tag{14.6}$$

When WAL is used, modified pages are asynchronously written back to the database. Until they have been written back, they are locked in the buffer. During one checkpoint interval, N_P pages in the current partition are updated, and we approximate the average number of locked pages to $N_P/2$, and the fraction of the pages in the buffer that is locked is $\frac{N_P/2}{S_{DB}/S_P}$ (except for very low buffer sizes, or large checkpoint intervals, this fraction will be very low). The average cost of reading a page when taking the buffer into account is:

$$T_{readpage} = (1 - P_{buf_opage})T_P \tag{14.7}$$

Assuming a clustering factor of C, the average object retrieval cost from the current partition is:

$$T_{readobj_cur} = \frac{1}{CN_{o_page}} T_{readpage}$$

The average object retrieval cost from the historical partition, where we can not assume any clustering is:

$$T_{readobj_hist} = T_{readpage}$$

We denote the probability that a read operation is for the current version as P_{RC}. The average object retrieval cost is:

$$T_{readobj} = P_{RC}T_{readobj_cur} + (1 - P_{RC})T_{readobj_hist} \tag{14.8}$$

It is difficult to benefit from compression of small objects when reading, because the whole page has to be read in any case.

[6] We do not consider the use of dual-buffering in this comparison, as its main application is in the context of client side buffering.

14.6.4 Object Update Cost

N_{CR} of the written objects during a checkpoint interval are creations of new objects:

$$N_{CR} = P_{new} N_{CP}$$

Of the objects written during a checkpoint interval, $(N_{CP} - N_{CR})$ are updates of existing objects, and the number of *distinct* updated objects is:

$$N_{DU} = N_{distinct}(N_{CP} - N_{CR}, N_{obj})$$

With a sufficiently large checkpoint interval, there will be several writes to a page. The average number of times each object is updated is:

$$N_U = \frac{N_{CP} - N_{CR}}{N_{DU}}$$

During one checkpoint interval, the number of pages in the current partition of the database that are affected is:

$$N_P = \frac{N_{DU}}{N_{o_page} C}$$

This means that during one checkpoint interval, new versions must be inserted into N_P pages. CN_{o_page} objects on each of these pages have been updated, and each of them has been updated an average of N_U times. To maintain the constraint in the model that all affected pages during one checkpoint interval fits in the buffer, N_P must always be less than the number of pages that fits in the buffer.

Not all of the objects in a temporal ODBMS are temporal. We denote the fraction of the object writes done to temporal objects as $P_{write_temporal}$. Only updates of these objects need to be written to the historical partition. For each of the N_P pages, we need to write $P_{write_temporal} N_U C N_{o_page}$ objects to the historical partition (this includes updated objects on the page and objects who was not installed into the page before they became historical), install the new current version to the page, and write it back. This will be done in batch, to reduce disk arm movement, and benefit from sequential writing of the historical objects. It is possible to compress historical objects before they are written to the historical partition. R_{comp} is the size of the compressed object relative to the uncompressed object, i.e., on average we manage to compress the objects down to R_{comp} of their original size. The cost of writing b bytes sequentially to disk, assuming that the amount of data is large enough to ignore the disk seek time:

$$T_S(b) = \frac{b}{V_s} t_r$$

When creating a new object, a new page will, on average, be allocated for every N_{o_page} object creations. We assume that these pages can be written efficiently, in a mostly sequential way. The total cost related to object write during one checkpoint interval is:

$$\begin{aligned} T_T &= \textit{Write to historical partition} \\ &\quad + \textit{Write modified pages} \\ &\quad + \textit{Write new pages} \\ &= T_S(N_P P_{write_temporal} N_U C N_{o_page} S_{obj} R_{comp}) \\ &\quad + N_P T_P \\ &\quad + T_S(\tfrac{N_{CR}}{N_{o_page}} S_P) \end{aligned}$$

Function	Definition	Equation/ Default Value
N_{RA}	Number of objects from each disk-read	1
P_{buf_obj}	Probability of finding an object in the buffer	(14.10)
S_{seg}	Segment size	512 KB
U_{objbuf}	Memory utilization in object buffer	0.9

Table 14.5: LO-ODBMS specific parameters and functions.

The average object update cost:

$$T_{writeobj} = \frac{T_T}{N_{CP}} \tag{14.9}$$

It is interesting to note the low cost of keeping old versions. If we are willing to pay the cost of the disk space, the most important additional cost will be the OIDX cost. The writing of old versions to a historical partition is cheap, compared to the writeback of modified pages.[7]

14.7 Analytical Modeling of an LO-ODBMS

The server modeled in this chapter is a server operating according to the description of the Vagabond ODBMS as previously described in this thesis. Some possible optimizations, for example signatures and delta objects, are not considered.

14.7.1 Object Read Cost

Similar to the IPU-ODBMS, we need two disks to be able to handle media failures. In a log-only system, we can use the disks in a mirrored configuration, which doubles the read bandwidth. The average cost of reading an object from disk when we have parallel, independent read operations from the two disks:

$$T_{readobj_disk} = (T_B(S_{obj}R_{comp}))/2$$

When reading an object, the object may already be resident in the object buffer. The memory utilization in an object buffer will be less than the memory utilization in a page buffer, because of variable length objects. We assume a memory utilization of U_{objbuf}. When the log-only approach is used, we do not need to lock objects after they have been written to the log, and we assume that the number of locked objects (for example dirty objects from ongoing transactions) is small enough to ignore. The probability of an object being resident in the object buffer is P_{buf_obj}:

$$P_{buf_obj} = P_{buf}(\frac{U_{objbuf}M_{buf}}{S_{obj} + S_{overhead}}, N_{obj}) \tag{14.10}$$

When we take the object buffer into account, the average object read cost is:

$$T_{readobj} = (1 - P_{buf_obj})T_{readobj_disk} \tag{14.11}$$

[7]With the default values used later in this analysis, the value of $T_{writeobj}$ with $P_{write_temporal} = 1.0$ is only a few percent higher than $T_{writeobj}$ with $P_{write_temporal} = 0.0$.

The assumption that only one object is read from the disk between each disk seek is actually a very conservative estimate. It does not take into account prefetching, and more importantly, it does not take into account disk read-ahead. In practice, dynamic reclustering will improve read performance considerably. Without doing a thorough analysis of this aspect, we can get an idea of how much we can gain from this by using the previous cost functions, and assuming that from each random read operation to retrieve an object, we can get N_{RA} additional objects "cheap". Cheap in this context assumes that the disk employs read-ahead, so that when we do a read operation, some of the following data will also be read. If doing a disk-read shortly after, this data will still be in the cache on the disk, and can be retrieved without an additional physical read operation. In this way, the ODBMS does not even have to do any implicit prefetching. If we assume N_{RA} to be small, so that the total data $D = N_{RA}S_{obj}$ is small enough to expect it to be cached by read-ahead, we can approximate the read cost in this case to be:

$$T_{readobj_reclustered}(N_{RA}) = T_{readobj}/N_{RA} \tag{14.12}$$

In this analysis we will in general assume the worst case, with $N_{RA} = 1$, but we will in Sect. 14.8.3 show how a more realistic value of N_{RA} will affect performance.

14.7.2 Object Update Cost

Writing to the log is sequential, with large blocks (segments). The object write operation in a LO-ODBMS is to write the object together with its OD to the log:

$$T_{writeobj} = T_S(S_{od} + S_{obj}R_{comp}) \tag{14.13}$$

Writing the OD together with the object avoids synchronous updates of the OIDX, because in an LO-ODBMS we do not have a separate log in which to write this information.

14.8 A Comparison of Performance

In this section we will use the cost functions to compare the performance of the LO- and IPU- approaches with different workloads and predicted access patterns. Access costs are dependent of the mix of read and write operations, and they must be studied together. If we denote the probability that an operation is a write, as P_{write}, the average access cost is the average of the cost of all object read and write operations:

$$T_{access} = (1 - P_{write})T_{readobj} + P_{write}T_{writeobj} \tag{14.14}$$

We will also use speedup as a metric in the comparisons. If we denote the average object access time for an IPU-ODBMS as T^{IPU}_{access}, and the average object access time for an LO-ODBMS as T^{LO}_{access}, we can calculate speedup as:

$$Speedup = \frac{T^{IPU}_{access}}{T^{LO}_{access}} \tag{14.15}$$

A speedup less than 1.0 means that with the given parameters, an IPU-ODBMS will perform best. A speedup greater than 1.0, means that an LO-ODBMS will perform best. In the figures, we have also plotted the 1.0 line to make it easy to see under which conditions each of the two approaches have the best performance. The memory size in the figures is the memory size relative to the total database size, i.e. M_{obuf}/S_{DB}.

14.8.1 Workload Model

Unless otherwise noted, results and numbers in the following sections are based on calculations using the default parameters, as summarized in Tables 14.3, 14.4, and 14.5. The size of the database to be accessed by the transactions is S_{DB}, excluding overhead. The number of object versions is N_{obj}. Given a fixed size of the database, N_{obj} is a function of the object size. Note that even though some of the parameter combinations in the following sections are unlikely to represent the average over time, they can occur in periods, for example, more write than read operations. It is in situations like this that adaptive, self-tuning systems can be very beneficial, when parameter sets differ from the average, against which systems have been tuned.

In this analysis, we study the performance using the four access patterns which were summarized in Table 14.1.

14.8.2 Performance in the Recovery and Heating Phases

In many application areas, it is important to deliver high availability. Low failure rates and fast crash recovery are necessary to achieve this. The duration of this phase is approximately a function of the amount of log that has to be processed, and the number of passes the log has to go through.

A log-only system, such as the one presented here, is able to provide fast crash recovery. Only one pass through the log written since the penultimate checkpoint is necessary. Thus, this phase can be of very short duration for a log-only DBMS. Traditional systems, employing in-place updating with WAL, need more than one pass over the log (although the log in those systems in general will be smaller than in a log-only system because logical logging can be used).

In the heating phase, a lot of buffer misses will be experienced. With the log-only approach, we do not necessarily have the same amount of clustering as in the traditional approach, and it is possible that more data have to be read from disk. In many cases, the real performance will not be too bad, as many of the hot-set objects will have been written in the part of the log read during recovery. This is also the case for the IPU-ODBMS.

14.8.3 Performance During Normal Operation

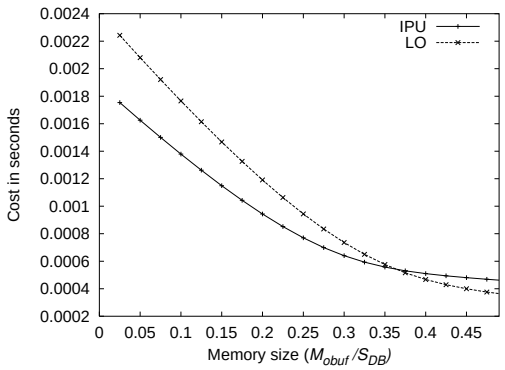
In this section, we compare the performance of the LO- and IPU-approaches during the normal operation phase, with different workloads and access patterns.

Object Access Cost

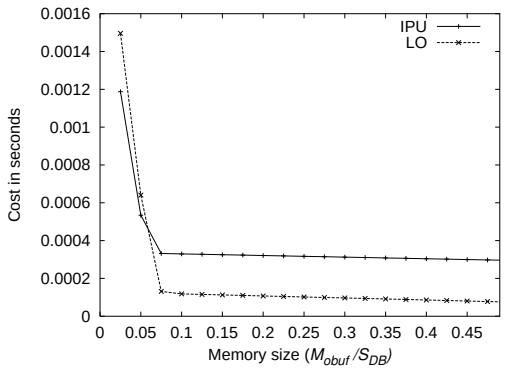
Figure 14.12 shows the average object access costs with different access patterns. We see here that the access pattern determines which approaches perform best. The bottleneck in an LO-ODBMS is the object retrievals, and figure shows that when we have sufficient memory to buffer the hot-spot objects, the LO-ODBMS will outperform the IPU-ODBMS.

The Effect of Different Object Sizes

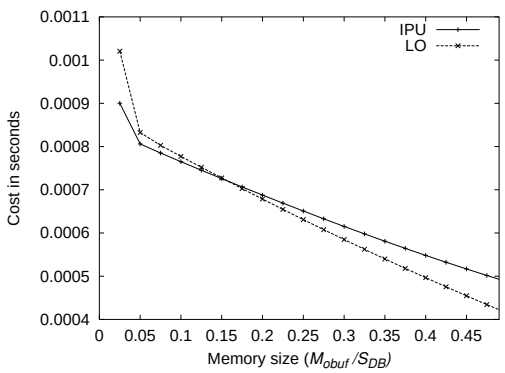
The average object size is an important parameter. In this study, we used a default object size of 208 bytes, which is close to size which we have observed used in many other ODBMS performance studies (our chosen value, 208 bytes, is the tuple size in the well known Wisconsin Benchmark for RDBMSs).



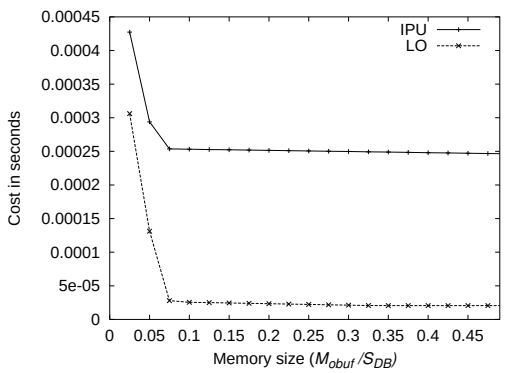
(a) 2P8020 access pattern.



(b) 2P9505 access pattern.

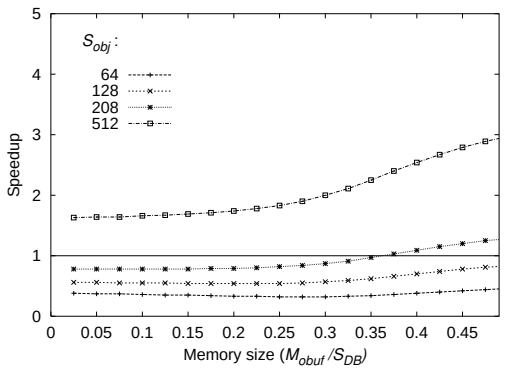


(c) 3P1 access pattern.

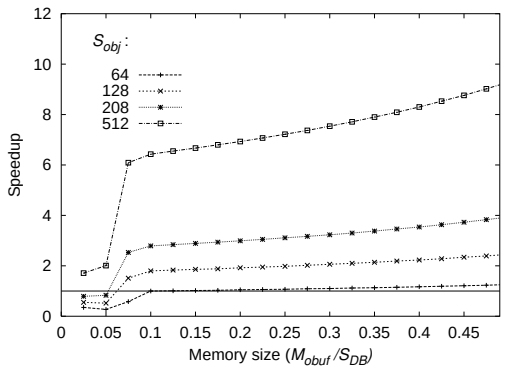


(d) 3P2 access pattern.

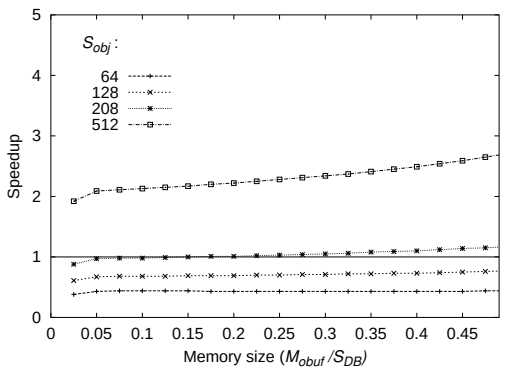
Figure 14.12: Object access cost with object size $S_{obj} = 208$.



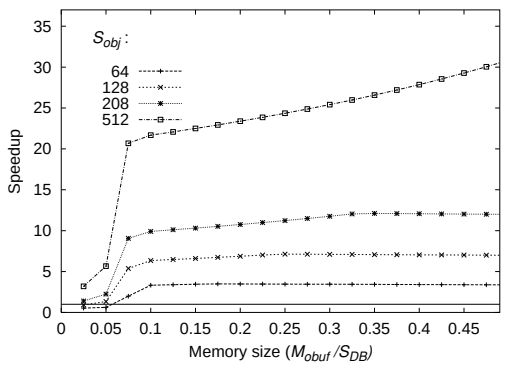
(a) 2P8020 access pattern.



(b) 2P9505 access pattern.

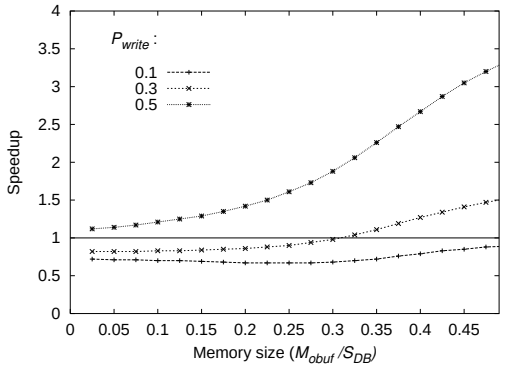


(c) 3P1 access pattern.

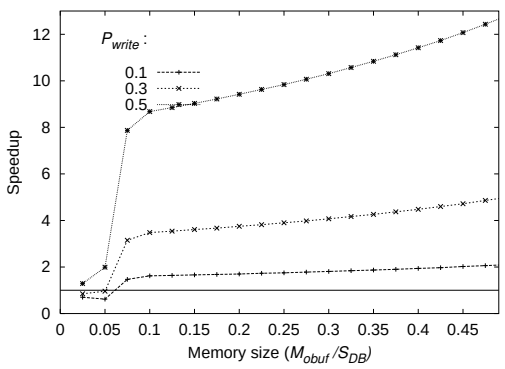


(d) 3P2 access pattern.

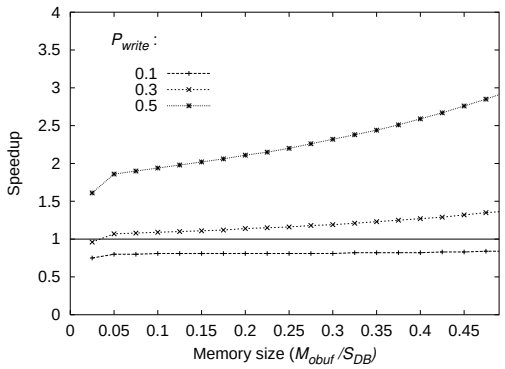
Figure 14.13: Speedup with different object sizes S_{obj}.



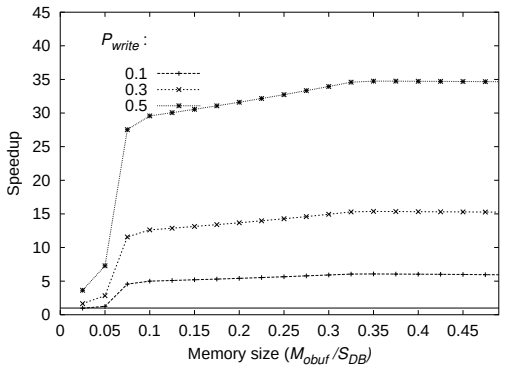
(a) 2P8020 access pattern.



(b) 2P9505 access pattern.



(c) 3P1 access pattern.



(d) 3P2 access pattern.

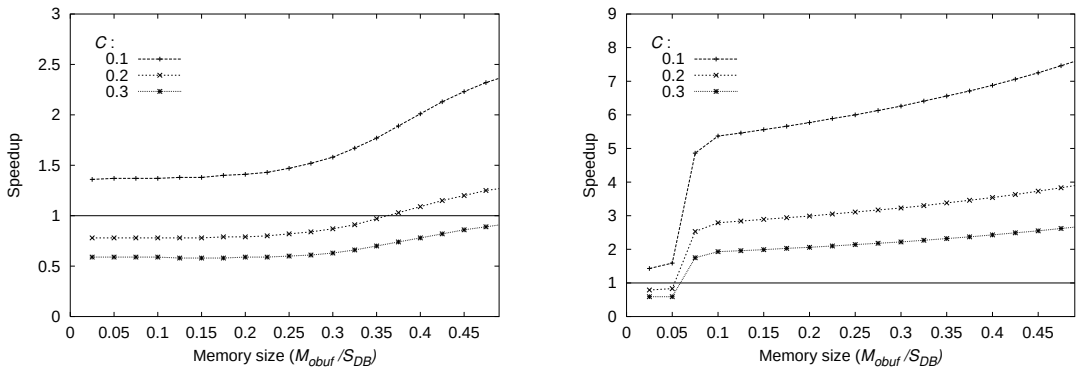
Figure 14.14: Speedup with different update ratios P_{write}.

Figure 14.13 illustrates the speedup from using a log-only based ODBMS with different object sizes. Larger objects reduces the impact of the seek time, which benefits an LO-ODBMS. The figure shows this effect very well.

In Figure 14.12 we saw how the average object access cost was significantly reduced when we had enough buffer memory to keep the most frequently accessed objects in memory. Figure 14.13 illustrates this even better, and we see a very significant gain from using a log-only based ODBMS.

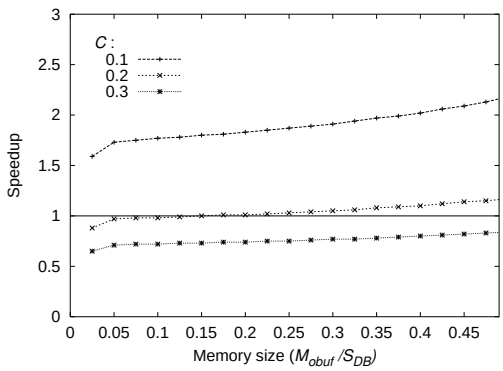
The Effect of Different Update Ratios

In this analysis, we have used $P_{write} = 0.2$ as the default value for the fraction of operations which are write operations. In some periods, and in some application areas, we will have a higher value of $P_{write} = 0.2$. Figure 14.14 shows how different update ratios affect the speedup. We see that with a higher value of P_{write}, the speedup is considerably higher than for small values of P_{write}. The reason is again that the read operation is the bottleneck in the LO-ODBMS, and a higher write rate reduces the average object access time.

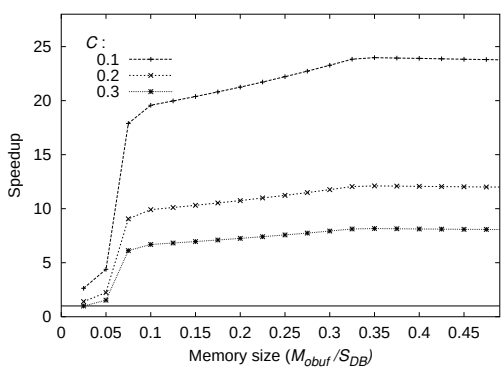


(a) 2P8020 access pattern.

(b) 2P9505 access pattern.



(c) 3P1 access pattern.



(d) 3P2 access pattern.

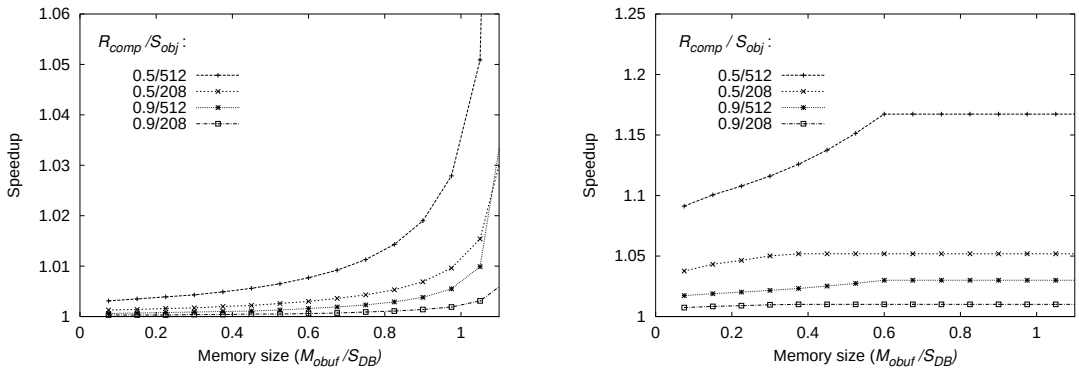
Figure 14.15: Speedup with different clustering factors C.

The Effect of Different Clustering Factors

In most page-server ODBMSs, it is possible to advise the system on how to cluster objects on the pages. Unfortunately, in most systems only the initial clustering can be specified, and reclustering is often impossible. In a multiuser system, with complex and dynamic workloads, it is commonly difficult to find a good clustering. The default clustering factor in our analysis is 0.20, which favors the IPU-ODBMS approach.

Figure 14.15 shows how the clustering factor (see Section 14.6.1) in IPU-ODBMSs affects their performance, and their relative performance to LO-ODBMS. This shows well how much IPU-ODBMSs depend on good clustering. This is an important point. In practice, with different applications accessing a database, it is difficult to achieve a high clustering factor. In a study by Tsangaris and Naughton [208], all practical clustering algorithms resulted in an average clustering which is less than this value, for the clustering algorithms and workloads in this study, C had values between 0.25 and 0.1.[8] In a real world application, clustering will be worse. That study was done with a 4 KB page

[8]Tsangaris and Naughton use the metric *expansion factor (EF)*, where $EF = 1/C$.



(a) 3P1 access pattern.

(b) 3P2 access pattern.

Figure 14.16: The speedup from using compression in an LO-ODBMS, with object sizes $S_{obj} = 208$ and $S_{obj} = 512$ bytes, and compression ratios $R_{comp} = 0.5$ and $R_{comp} = 0.9$.

size. With a larger page size, C would decrease because of a higher degree of false sharing.

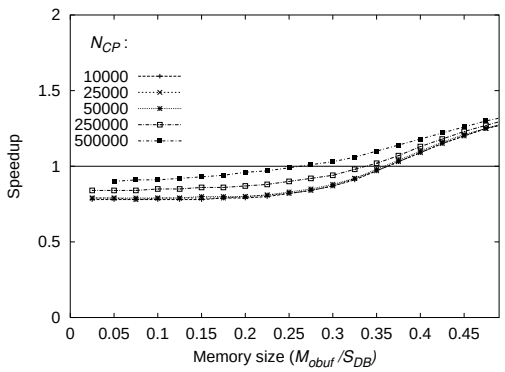
As is evident for the figure, if we consider a more likely clustering factor, less than 0.2, an LO-ODBMS will perform better than an IPU-ODBMS for both access patterns, even with a much smaller amount of main memory available.

The Effect of Different Compression Ratios

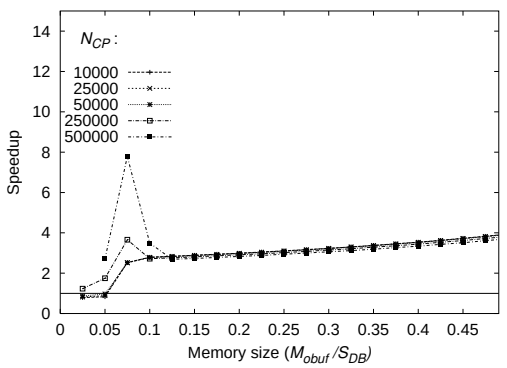
By compressing the objects, we can reduce the amount of data that has to be transferred between main memory and the disk. For each of the approaches (IPU-ODBMS and LO-ODBMS), we have calculated the speedup from using compression compared to not using compressions, i.e., $Speedup = \frac{T_{access}(R_{comp}=1.0)}{T_{access}(R_{comp}<1.0)}$.

Figure 14.16 shows how much we can gain from using object compression in an LO-ODBMS in the case of relatively small object (208 and 512 bytes). For small objects, the amount of compression rate will in general be quite small, and we have in this study used compression ratios of $R_{comp} = 0.9$ and $R_{comp} = 0.5$. The figures shows that with these parameters, an LO-ODBMS gets moderate speedup. By using compression adaptively, for example on certain object classes or on large objects, the speedup should be high enough to make compression beneficial. However, it also shows that with small objects, the cost of disk seek when retrieving an object from disk is very significant, and only when most accessed objects are resident in the object buffer (i.e., a large object buffer or a small hot-spot partition), the data volume itself will be a significant factor for the total cost.

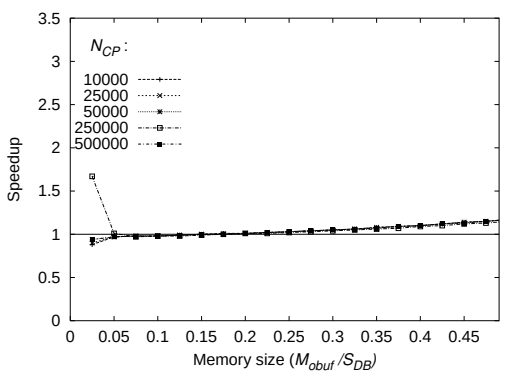
In the case of an IPU-ODBMS, the speedup was not significant (less than 1.01). Only objects written to the historical partition are compressed, and as the writing to the historical partition had only little impact on the total cost, this results in little gain from using compression. However, with larger objects, there will be more to gain from compression in both IPU-ODBMSs and LO-ODBMSs.



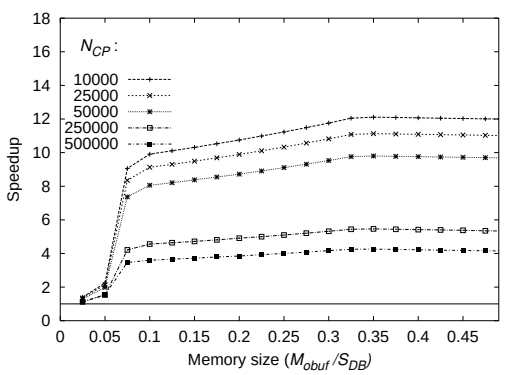
(a) 2P8020 access pattern.



(b) 2P9505 access pattern.

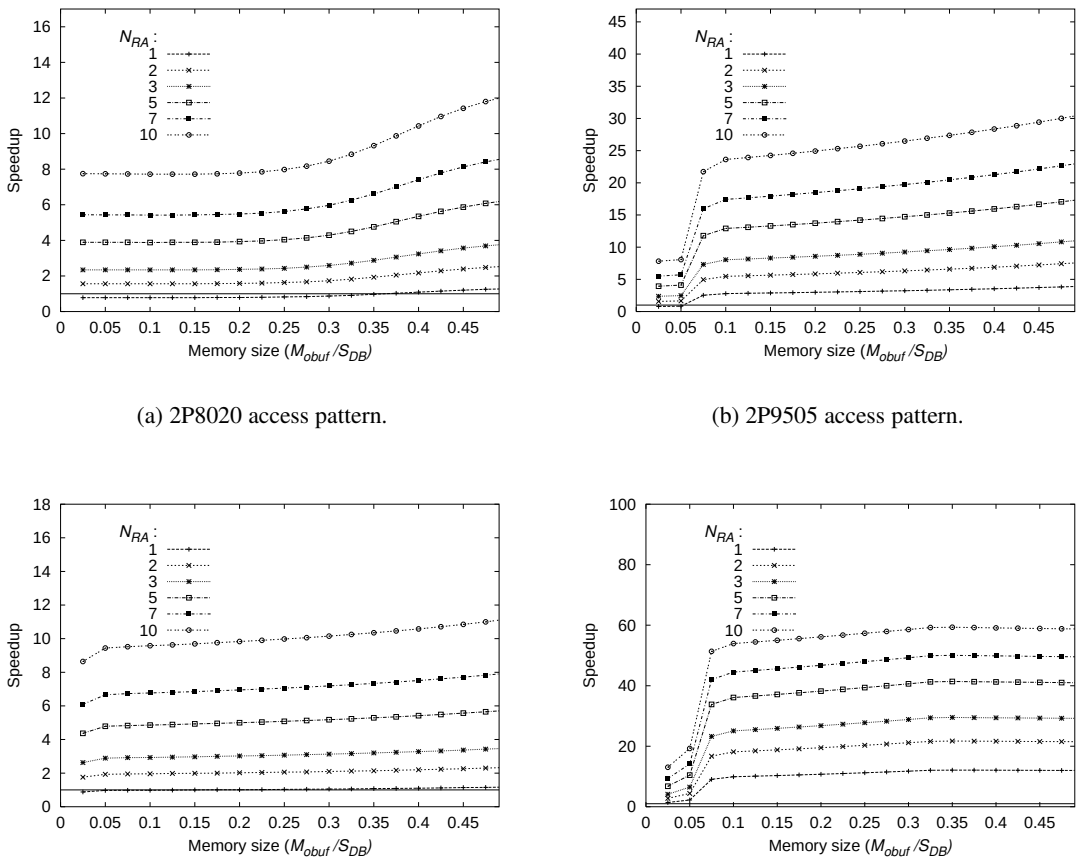


(c) 3P1 access pattern.



(d) 3P2 access pattern.

Figure 14.17: Speedup with different checkpoint-interval lengths N_{CP}.



(a) 2P8020 access pattern.

(b) 2P9505 access pattern.

(c) 3P1 access pattern.

(d) 3P2 access pattern.

Figure 14.18: Speedup with disk read-ahead.

The Effect of Different Checkpoint-Interval Lengths

The checkpoint-interval length N_{CP} is a tradeoff between performance, recovery time, and page buffer hit rate (reduced writeback rate locks more pages in the buffer). We have until now used a default value of $N_{CP} = \frac{2\,\text{MB}}{S_{obj}}$ objects, i.e., $\approx$ 10000 new objects (object versions) between each checkpoint. In Figure 14.17 we illustrate how the speedup from using an LO-ODBMS changes with different checkpoint-interval lengths in the modeled IPU-ODBMS. We see that an increased checkpoint-interval length is particularly beneficial for an IPU-ODBMS when we have a very small hot set. In this case, there are many modification to the page before it has to be written back to disk. In other cases, there is less to gain from a large checkpoint-interval length. This is particularly evident when it results in too many locked items in the buffer, which reduces the hit rate. This can be seen on the plots for the 2P9505 access pattern.

The Effect of Clustering and Disk Read-Ahead in an LO-ODBMS

Until now, we have assumed no clustering in the LO-ODBMS, with only one object read on each disk access. It is very likely that it is possible to achieve better than this. This can be achieved by writing related objects at the same time, and by "intelligent" cleaning and reorganization of the segments.

By using a relatively high default value for the clustering factor in an IPU-ODBMS, we have already assumed it is well clustered. By using the modified LO read object cost in Equation 14.12 with different values of N_{RA}, we get an indication of how much can be gained. Figure 14.18 shows speedup with different values of N_{RA} with the different access patterns.

14.8.4 The Impact of Cleaning and Reorganization

Real performance will depend on the cost of the cleaning and reorganization.

Cleaning Cost

The cleaner runs in the background, and employs idle resources. However, under certain conditions, its cost will be significant:

- In a system with no idle periods, the disk will either fill up because of no cleaning, or we have to make the cleaner compete with normal operations. Performance degradation will be experienced
- With high disk utilization, and little idle time, segments do not have time enough to let their data become "dead". In this case, the cleaner will have to process many nearly full segments to get enough free space for one new segment. This can be costly.

Cleaning cost is an important issue that should be investigated further. However, the cleaning can also cluster related objects together, and we hope the benefits from this reclustering will outweight the disadvantages of the cleaning process itself.

Reorganization

Reorganization is an important (too important!) issue in an IPU-ODBMS. To be able to benefit from object clustering, the database clustering has to change according to the change in changing applications and access patterns.

Traditionally, reorganization had to be done by dumping and restoring the database, and in many systems, this is still the only way to do it. Most ODBMSs capable of handling large amounts of data use physical OIDs, and in such systems, online reorganization is difficult. This affects availability as well as performance.

14.9 OIDX Costs

In order to have a balanced system, the OIDX-access costs must match the object-access costs. If we compare the object-access costs with the OIDX-access costs in the paper in Appendix D, we see that the OIDX-access costs in that case are much higher. For example, in the case of the 2P9505 access pattern, the OIDX-access costs are 50 times higher. One of the reasons for this difference, is that

we use parameters for faster disks in the study of the object-access costs than in the study of OIDX-access costs (the paper in Appendix D was written in 1998). However, even when using the same disk parameters, the OIDX-access costs are 30 times higher. To compensate for this cost, a large number of disks for OIDX storage would be needed, but fortunately, we expect the difference between object and OIDX-access cost to be much lower in practice:

1. The study in Appendix D is based on a composite index (see Section 8.3.2), with little locality of accesses. By using the VTOIDX, the locality will be much higher, because the ODs of current versions are separated from ODs of historical versions, and the ODs from a container are clustered together. This means that the number of random read accesses is reduced.

2. The use of the PCache will further reduce the OIDX-access cost.

14.10 Summary

We have in this chapter used analytical cost models to compare the expected performance of temporal ODBMSs based on the log-only and in-place update approaches. The most important part of these models, the buffer models, was validated by comparing the models with simulation results.

In order to make the model numerically tractable, some simplifying assumptions were necessary. For example, the application areas characterized statistically in the analysis overlooks the coherence and structure of actual applications, which affects the accuracy of the cache behavior prediction.

The initial exploration of the relative performance of the log-only and in-place update implementation strategies suggests that there is a class of applications for which log-only may be beneficial. Further work is required to establish the extent of this class and the actual benefits obtained. We believe such further investigation is warranted.

Chapter 15

A Comparison of Declustering Strategies

In this chapter, we do a simple qualitative analysis of the declustering strategies discussed in Chapter 11. We develop cost models for the declustering strategies, and then do an analysis of the performance of the declustering strategies.

15.1 Cost Model

In the analysis, we assume a database system executing on a number of nodes communicating through a message-based communication network. An instance of the database system runs on each node. On each node, a number of objects are stored. In general, a client is connected to one of the nodes. Simple queries can be run on the same node as that to which the client is connected, more complex queries are parallelized and run in parallel on several nodes.

When objects are created or updated, they are stored on one of the nodes, according to one of the declustering strategies described in Section 11. The declustering strategy is used when retrieving objects, in order to be able to know on which node a certain object or object version is stored. Each node indexes the objects stored on that node. In addition to traditional object retrieval and object relational operations, temporal operations have to be supported.

The goal of our analysis is to compare the different declustering strategies in a system with N_N nodes, and investigate under which conditions they should be used. We use a qualitative approach, where our goal is to achieve a cost model of the bottleneck in such a system. We make the following assumptions:

1. In a balanced system, the CPU and disk costs will be the same for the different declustering strategies, hence, we only consider the communication costs.

2. We assume a high enough degree of intra- and inter transaction processing to be able to exclude response times in the processing from the cost model.

3. We assume that a number of messages and objects can be packed together in each packet, so that the number of messages in itself can be ignored.

Under these assumptions, the transported data volume of the objects and messages can be used as a measure. We denote the cost of sending an object (including the object as well as object identifying information) in terms of bytes from the communication bandwidth as C_S, and the of sending a message, for example a message requesting an object, as C_P. The parameters and functions used in the cost model is summarized in Table 15.1.

Parameter	Definition	Default Value
α	Fraction of accesses to hot spot objects	0.95
β	Fraction of total # of objects that are hot spot objects	0.05
C_P	Cost of sending an *OID/TIME* object probe msg.	16 B
C_{PP}	Cost of sending an *OID/TIME/TIME* object probe msg.	24 B
C_S	Cost of sending an object (including overhead)	256 B
N_N	Number of nodes	-
P_{write}	Object write probability	0.2
P_{new}	Probability that a write creates a new object	0.2
P_{RC}	Probability that a read is for the current version	0.7
P_{RH}	Probability that a read is for a historical version	0.1
P_{TS}	Probability that a read is a timeslice operation	0.2

Function	Definition
C	Average cost of an object access
C_C	The average cost of an object create operation
C_R	The average cost of a read operation
C_{RC}	The average cost of reading a current object version
C_{RH}	The average cost of reading a historical object version
C_{TS}	The average cost of a timeslice read
C_U	The average cost of an update operation
C_W	The average cost of a write operation
P_{RU}	Probability that an object has been updated during the current time range

Table 15.1: Summary of system parameters and functions.

Not all the objects in a temporal ODBMS are temporal, for some of the objects, we are only interested in the current version. To improve efficiency, the system can be made aware of this. In this way, some of the objects can be defined as non-temporal, and old versions of these are not kept. Access and declustering of these objects are independent from temporal objects, and in this analysis we only consider access to and declustering of the temporal objects.

15.1.1 Workload Model

The workload consists of object updates and read accesses. In order to study the performance of the different declustering strategies, it is important to study the costs of the updates and read accesses together. In the modeled workload, P_{write} of the accesses are object updates, and of these, P_{new} are creations of new objects. $(1 - P_{write})$ of the accesses are accesses caused by read accesses/read queries. Denoting the average write and read costs as C_W and C_R, we calculate the cost as the average bandwidth needed for each object access operation, i.e.:

$$C = P_{write} C_W + (1 - P_{write}) C_R$$

The average cost of a write is the weighted sum of the average object create cost C_C and the average object update cost C_U:

$$C_W = P_{new} C_C + (1 - P_{new}) C_U$$

The read pattern from different applications can be divided into several categories, for example:

- Applications only accessing current versions.
- Applications that mostly accessing current version, and a few historical versions through navigations.
- Applications that do timeslice operations, i.e. operating on a snapshot valid at time T_i.
- Set accesses (queries) only accessing current versions.
- Set-based accesses involving temporal operators.

The object accesses from these different application categories can be incorporated into 3 read classes, each representing a certain fraction P_i of the read accesses, where the invariant $P_{RC} + P_{RH} + P_{TS} = 1.0$ should be true:

1. P_{RC}: Read the current version of an object. The cost of this operation is C_{RC}.
2. P_{RH}: Read a historical version of an object, i.e., an object version valid at a particular time T_i. This category of read operations also includes the case when we want to retrieve all object versions of a particular object. The cost of this operation is C_{RH}.
3. P_{TS}: Timeslice read operations, which can benefit from a declustering strategy that aims at storing object versions valid at the same time on the same node (see Section 11.1). The cost of this operation is C_{TS}.

The average cost of a read is:

$$C_R = P_{RC} C_{RC} + P_{RH} C_{RH} + P_{TS} C_{TS}$$

We will now present the cost models for the *OID*, *TIME*, and *OID-TIME* declustering strategies.

15.1.2 *OID* Declustering

When an object is created, the node where it is to be stored is determined from applying a hash function to the OID. With a probability of $\frac{1}{N_N}$, this is the same node as the creating node. If it is another node, which has the probability $(1 - \frac{1}{N_N})$, the new object has to be sent to the actual node. This also applies to object updates. The average object create and update costs are:

$$C_C = C_U = (1 - \frac{1}{N_N})C_S$$

All versions of an object reside on the same node, so that the costs of reading current and historical versions, as well as the cost of doing a timeslice read, are the same. If the requested object version is not stored on the requesting node (the node which does the operation), the average access cost is the sum of the cost of the object retrieve message and the cost of returning the object:

$$C_{RC} = C_{RH} = C_{TS} = (1 - \frac{1}{N_N})(C_P + C_S)$$

15.1.3 *TIME* Declustering

When an object is created, the node where it is to be stored is determined from the value of its timestamp. With a probability of $\frac{1}{N_N}$, that is the same node as the creating node, and no communication is necessary. If it is another node, which has the probability $(1 - \frac{1}{N_N})$, the new object has to be sent to the actual node. The average create cost is:

$$C_C = (1 - \frac{1}{N_N})C_S$$

When an object is updated, the new object version is sent to the node determined from the timestamp. In addition, a message is sent to the node where the previous current version was stored, so that the end timestamp can be set for the previous current version (this reduces the current version access cost):

$$C_U = (1 - \frac{1}{N_N})C_S + (1 - \frac{1}{N_N})C_P$$

To read the current version of an object can be a very expensive operation when *TIME* declustering is used. All nodes have to be probed if the current version is not stored on the requesting node. The cost is the sum of sending a probe message to all other nodes, and sending the requested object back to the requesting node (the node that stores the current version can determine this from the end timestamp, which is not set in the current version):

$$C_{RC} = (1 - \frac{1}{N_N})((N_N - 1)C_P + C_S)$$

An object is valid in a given time interval, from the value of the timestamp, and until the timestamp of the next object version. This interval covers one or more ranges in the range partitioning, including one or more nodes.

We assume an access pattern with a small number of frequently updated objects, where α of the updates are performed to these objects. We assume that the frequently accessed objects are updated often enough to be updated more than once during each time range. The infrequently updated objects are updated very seldom, so that each version covers more than N_N time ranges. This is a

simplification, but the following example shows that it is not unreasonable: Consider a database in a stable condition, with size $S_{DB} = 32$ GB of data and average object size $S_{obj} = 256$ bytes, giving $N_{ver} = S_{DB}/S_{obj} = 128$M object versions. If we assume the number of time ranges to be $N_R = 128$, the number of object versions in each time range is $N_{objver}/N_R = 1$M versions. During each time range, $P_{new} = 0.2$ of the object versions are new objects, while $(1 - P_{new}) = 0.8$ are new object versions of existing objects. The number of distinct objects (i.e., distinct OIDs) in the database when in a particular time range i is approximately $N_d = P_{new} N_{ver} \frac{i}{128}$ objects, on average $P_{new} N_{ver} \frac{1}{2} = 12.8$M objects. Under the assumptions above, $\alpha = 0.95$ of the updates are done to $\beta = 0.05$ of the N_d objects, this means that α of the updates are done to $\beta N_d = 0.64$M objects, i.e., ≈ 1.5 updates of each hot spot object in each time range.

When requesting the version of an object that was valid at time T_i, we first send a probe message to the node which stores objects with timestamps in the time range which includes T_i (with a probability of $\frac{1}{N_N}$ this is the requesting node, and it is not necessary to send the probe message). If we assume the read pattern is equal to the write pattern,[1] the probability that the requested object version is stored on this node is $> \alpha$. Given a particular time in a time range, the probability that one of these objects has already been updated in this time range is 0.5, and with a probability of 0.5 we have to probe the previous node. The average cost of retrieving one of these object versions is:

$$\begin{aligned} C'_{RH} \;=\; & (1 - \tfrac{1}{N_N})C_P && \text{Send probe message} \\ & +0.5(1 - \tfrac{1}{N_N})C_S && \text{Return object} \\ & +0.5(C_{PP} + (1 - \tfrac{1}{N_N})C_S) && \text{Forward probe message} \\ & && \text{which returns object} \end{aligned}$$

We make a pessimistic assumption about the less frequently updated objects, and assume the average time between each update is larger than N_N time ranges. Thus, the probe message has to be forwarded $(N_N - 1)$ times. The average cost of retrieving a historical object version of one of the infrequently updated objects is:

$$\begin{aligned} C''_{RH} \;=\; & (1 - \tfrac{1}{N_N})C_P && \text{Send probe message} \\ & +(N_N - 1)C_{PP} && \text{Forward probe messages}^2 \\ & +((1 - \tfrac{1}{N_N})C_P + (1 - \tfrac{1}{N_N})C_S) && \text{Return object} \end{aligned}$$

The average cost of retrieving a historical object version is:

$$C_{RH} = \alpha C'_{RH} + (1 - \alpha) C''_{RH}$$

The timeslice read is essentially a read of a historical version on the node that covers the actual timeslice. However, we do not need to send the first probe message because we are already on the actual node covering the timeslice time, and if the object is stored on this node we do not have to send it:

$$\begin{aligned} C'_{TS} \;=\; & 0.5(C_{PP} + (1 - \tfrac{1}{N_N})C_S) && \text{Forward probe message} \\ & && \text{and return object} \end{aligned}$$

[1] As will be discussed later in this chapter, this assumption is possibly too optimistic, or even unlikely.

[2] Here we assume that all forward probe messages include *OID/TIME/TIME*. However, if the number of object versions of each of these objects is low, it is possible to reduce the communication cost by only including *OID/TIME* in the messages until the first node with a version of the object is reached.

$$\begin{aligned} C''_{TS} &= +(N_N - 1)C_{PP} && \text{Forward probe messages} \\ &\quad +((1 - \tfrac{1}{N_N})C_P + (1 - \tfrac{1}{N_N})C_S) && \text{Return object} \end{aligned}$$

$$C_{TS} = \alpha C'_{TS} + (1 - \alpha)C''_{TS}$$

As described in Section 11.3.2, using the same algorithm for retrieving current versions as historical versions can be beneficial. In that case we also avoid the need for maintaining the end timestamp, and the update cost is reduced to:

$$C'_U = (1 - \frac{1}{N_N})C_S$$

In the rest of this chapter, we will denote the approach where we use the same algorithm for retrieving current versions as historical versions as *TIME2* declustering.

15.1.4 *OID-TIME* Declustering

When creating and updating an object, the object is sent to the node where it is to be stored, based on the hash value of the OID. In addition, the object is sent to the node where it is to be stored based on the timestamp:

$$C_C = C_U = 2(1 - \frac{1}{N_N})C_S$$

When reading the current version of an object, we can avoid communication in two cases:

- If the requesting node is the same node as determined by hashing the OID, the current version is stored on the requesting node. The probability for this is $\frac{1}{N_N}$.
- The current version of an object is also stored on the node determined by its timestamp. If the requesting node is the one that covers the current timerange *and* the requested object was written during this time range. In that case, we know that there can not exist a more recent version on another node. The probability for this is $\frac{P_{RU}}{N_N}$, where $P_{RU} \approx \frac{\alpha}{2}$ is the probability that an object has been updated during the current time range. Note that if the current version of an object was written during a previous time range covered by the requesting node, we do not have enough information to know that this is still the current version the object. The reason for this, is that when *OID-TIME* declustering is used, we do not maintain the end timestamps for the objects.

In all other cases, communication is necessary. The average cost of reading the current version is:

$$C_{RC} = (1 - \frac{1}{N_N})(1 - \frac{P_{RU}}{N_N})(C_P + C_S)$$

The cost of retrieving a historical object version and the cost of a timeslice read is the same as when using *TIME* declustering.[3]

[3] It is possible that the current version of an object is the matching object in the case of retrieval of a historical object version or timeslice, but the probability of this is low enough to ignore.

15.1.5 Broadcasting

Efficient broadcasting/multicasting is often supported by the communication network. If this is the case, the cost of sending the same message to all the nodes can be significantly less than the cost of sending $_N$ messages. This can be utilized to reduce the number of probe messages when reading the current version using *TIME* declustering. Broadcasting can also be used when reading historical versions and doing timeslice read.

15.2 Analysis

The value of the parameters used in the cost model will be different from database to database, and even between different applications accessing a particular database. We will in this section study under which conditions the different declustering strategies are most beneficial, using a wide range of workloads and number of nodes.

The total communication cost increases with an increasing number of nodes, because the probability of finding an object on the same node decreases. In this analysis, we study the cost with different number of nodes, and the declustering goal is to minimize the communication costs given a certain number of nodes. We also want to study the scalability of the declustering strategies, i.e., that the increase in communication cost with increasing number of nodes is acceptable.

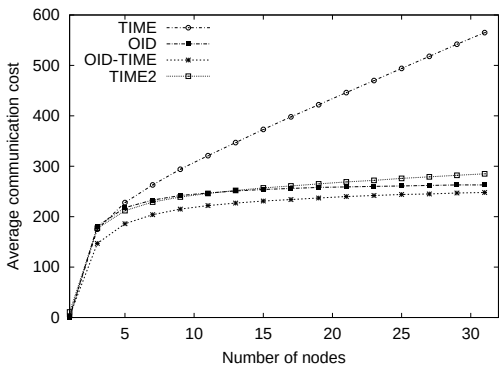
The Effect of Different Update Rates. Figure 15.1 illustrates the cost with different update rates. We see that the *OID-TIME* declustering is most suitable in the case of databases with low update rates. Typical examples of applications where this often occurs, are GIS (geographical information systems) and DSS (decision support systems). The cost of *TIME* declustering is high, because of the cost of retrieving current versions. The cost of *OID* declustering is low, predictable, and stable.

The Effect of Different Object Sizes. Different object sizes obviously affect the communication costs, and Figure 15.2 illustrates the cost with different object sizes. The performance of *OID-TIME* declustering gets worse with increasing object sizes because of the replication of the current object versions. The cost of using *TIME* and *TIME2* declustering is significantly lower than *OID* and *OID-TIME* declustering in the case of larger objects. The reason for this, is that the cost of the probe messages is less significant. However, it will not scale to a larger number of nodes. If the number of nodes is increased, this increases the number of probe messages and the total cost of *TIME* and *TIME2* declustering.

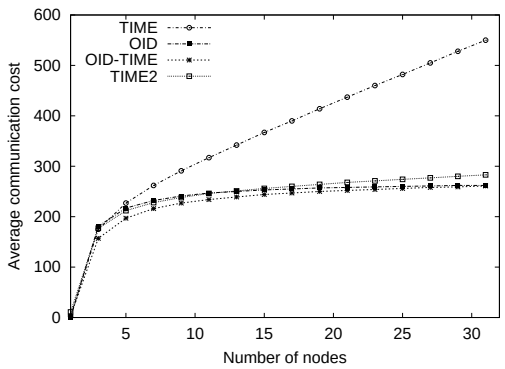
The Effect of Different Read Mix. Figure 15.3 illustrates the cost with different read mixes. We see how the performance of the *OID-TIME* and *TIME2* declustering increases relative to *OID* declustering with a larger amount of timeslice read operations. However, the amount of timeslice read operations needed to outperform the *OID* declustering is higher than what can be expected to occur in practice.

Scalability. From the figures, we have seen that the *TIME* declustering does not scale as well as *OID* and *OID-TIME* declustering. The main reason for this is that the navigational accesses to the current version objects are very expensive. Even with small values of P_{RC} this declustering strategy performs worse than the *OID-TIME* strategy.

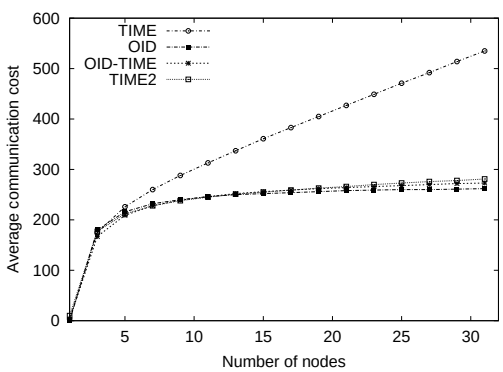
In the figures in this chapter, we have only illustrated the cost with up to 32 nodes. With a larger number of nodes, the cost of using *TIME2* declustering also becomes worse. A larger number of



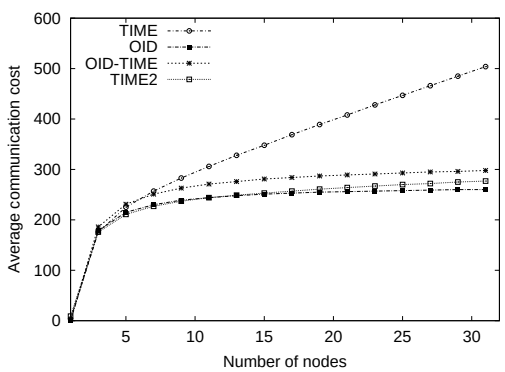
(a) $P_{write} = 0.0$.



(b) $P_{write} = 0.05$.

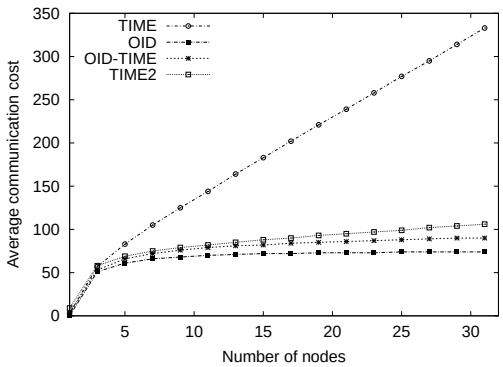


(c) $P_{write} = 0.1$.

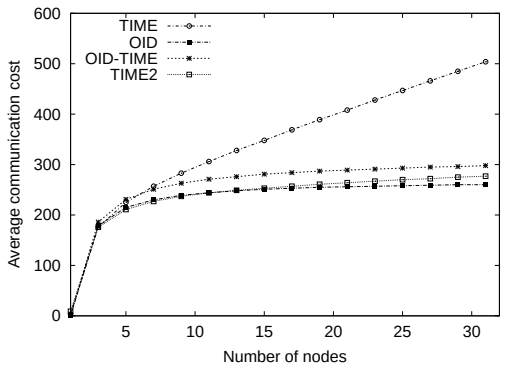


(d) $P_{write} = 0.2$.

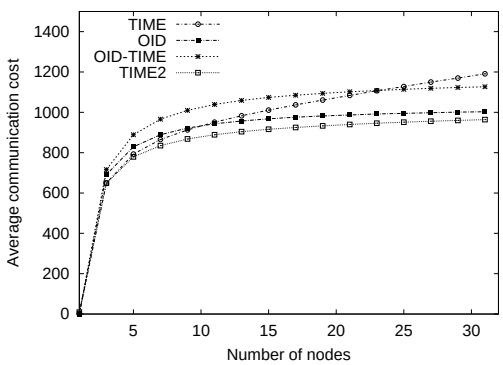
Figure 15.1: Cost with different update rates P_{write}.



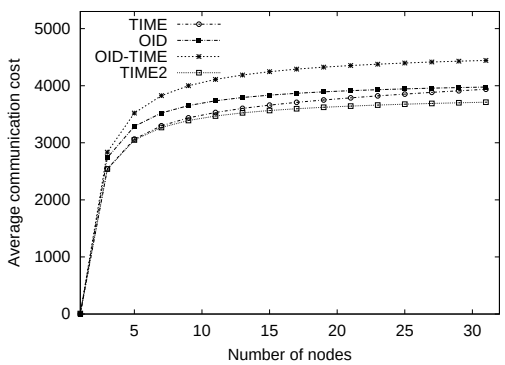
(a) $C_S = 64$.



(b) $C_S = 256$.

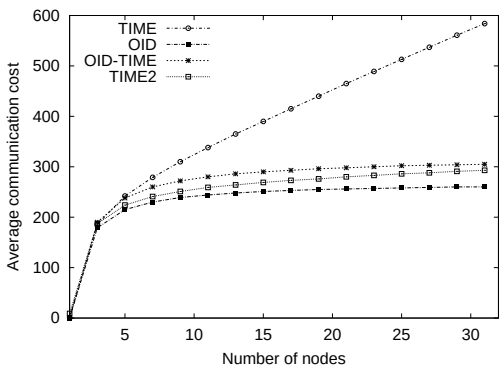


(c) $C_S = 1024$.

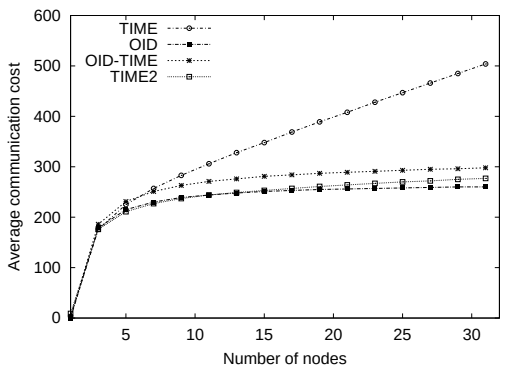


(d) $C_S = 4096$.

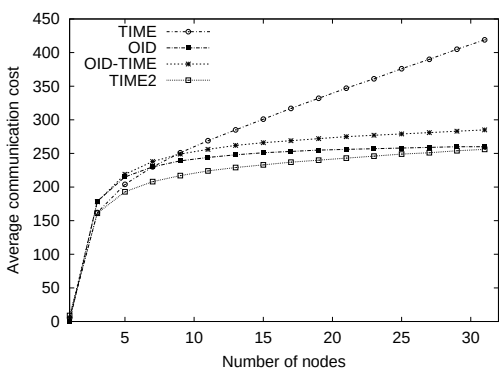
Figure 15.2: Cost with different object sizes.



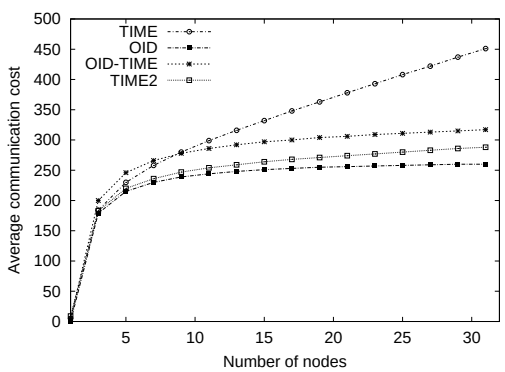
(a) $P_{RC} = 0.9$, $P_{RH} = 0.05$, and $P_{TS} = 0.05$.



(b) $P_{RC} = 0.7$, $P_{RH} = 0.1$, and $P_{TS} = 0.2$ (default mix used in the rest of the analysis).



(c) $P_{RC} = 0.5$, $P_{RH} = 0.1$, and $P_{TS} = 0.4$.



(d) $P_{RC} = 0.5$, $P_{RH} = 0.4$, and $P_{TS} = 0.1$.

Figure 15.3: Cost with different read mix.

nodes increases the number of times the probe messages have to be forwarded. This also increases the response time, also an important issue. A larger number of nodes also makes it necessary to use more time ranges in order to avoid skew. This is likely to further increase the cost of using *TIME* and *TIME2* declustering because it increases the probability of having to forward probe messages.

15.3 Summary

We have in this chapter analyzed strategies for declustering objects in parallel temporal ODBMSs. Cost models were developed and used to study the characteristics of the declustering strategies, and under which conditions the different declustering strategies are most beneficial. The results show that with the given assumptions:

1. In a parallel ODBMS, the *TIME* and *TIME2* declustering strategies are in general not suitable for primary declustering (but this does not rule out these strategies as a part of the later stages in the query processing).
2. In systems with mixed workloads, the difference in performance between the *OID* and *OID-TIME* strategies is not large, but in our choice of parameter values, *OID* declustering outperforms *OID-TIME* more often than the opposite.
3. Even in systems with a high degree of temporal operations, the cost of object traversal will be the most important cost.

It is also likely that the assumptions we made for the *TIME*, *TIME2* and *OID-TIME* declustering strategies were too optimistic:

1. The read pattern does not necessarily equal the write pattern.
2. Except some systems with mostly static data, for example data warehousing systems, a much larger number of time ranges will be used.
3. A low value of β (fraction of total # of objects that are hot spot objects) were implicitly assumed. A higher value of β would imply that each time range has to be larger, or a lower number of time ranges in total. In a more accurate model, this fact should be included in the model.

If any of these assumptions does not hold, the result will be a higher cost if using the *TIME*, *TIME2* or *OID-TIME* declustering strategies.

The final conclusion is that similar to traditional database systems, a hash-based declustering (*OID* based declustering) should be used in parallel temporal ODBMSs as well. However, we believe that the use of *OID-TIME* declustering during query execution is interesting, and this should be studied further.

Chapter 16

Conclusions and Further Work

In this chapter we summarize the thesis and its contributions, and outline issues for further research.

16.1 Is Vagabond a Suitable Solution?

We have in this thesis presented Vagabond, a temporal ODBMS based on the log-only approach. The main goal of this thesis has been to show that such a system is feasible, and this thesis is also an attempt to explore further into the "log-only space". We have identified some of the problems with the log-only approach, and proposed solutions to these problems. The main experience from this work is that designing a log-only system is one thing, designing an *efficient* log-only system is something completely different. The performance analysis does not give a definite answer to whether it should be the preferred solution, but the results are interesting enough to justify more research into this area.

In Section 1.1 we discussed several application areas that are only partially supported by current DBMSs. Our goal was to design a DBMS that could solve the problems better. In the rest of this section, we analyze how Vagabond solves the needs from these application areas.

16.1.1 Picture Archiving and Communications Systems

PACS is an application area that can benefit very well from using the log-only approach. Frequently, when a traditional DBMS is used, the only PACS data stored in the DBMS is picture describing data. The pictures themselves are stored using file systems external to the DBMS. Using Vagabond, it should be possible to achieve high performance even when all data is stored in the DBMS:

- The object modification rate in PACS will be very low, and pictures, for example X-ray pictures, will never be updated. The possibility of using large subobjects makes large object management efficient.

- The support for temporal object management in Vagabond will initially only be used as a versioning mechanism in PACS. However, temporal object support also makes it possible to perform queries that can help in finding relationships between patients and medical problems, and how these develop over time.

- PACS store large amounts of data (typically several TB). Most of the historical data in PACS systems will be very infrequently accessed and can be stored in tertiary storage. This is well supported in the Vagabond approach.

- As PACS will store large amounts of data, they are likely to be realized as parallel systems, with objects declustered based on OID (time-alignment operations will not be very frequent in a PACS application). In a telemedicine context distributed PACS are useful. Realized with Vagabond, this can mean server groups at each hospital, connected in a distributed system as outlined in Section 6.4.

16.1.2 Geographical Information Systems

Geographical information systems (GIS) is one of the application areas that can benefit most from the features provided by Vagabond:

- Complex real-world objects and relationships are well supported by the object model.

- Often, a GIS models the real world to a higher degree than most other applications, as it stores the physical appearance of the world around us. The world changes continuously, and by providing temporal object support it is easier to incorporate this into the GIS. However, Vagabond only supports transaction-time, and valid time has to be supported by including the valid-time interval in the objects and using additional valid-time indexes.

- The log-only approach is particularly well suited for applications such as GIS where objects are seldom updated. This reduces the need for segment cleaning as well as the amount of OIDX updates.

- A GIS should also be able to handle large objects. Even if a physical object, for example a house, often is stored in the database as a simple polygon, it is also desirable to be able to associate it with a picture or video of the house.

16.1.3 Scientific and Statistical Databases

The requirements of an SSDB listed in Section 1.1 could be well supported by Vagabond:

- Very large databases can be stored in Vagabond. Both objects and parts of the OIDX can be migrated to tertiary storage.

- The log-only approach is particularly well suited for applications where objects are seldom updated.

- In an SSDB, the low read/write ratio means that the amount of read operations will be relatively low. The number of distinct objects that will be read is low, and most of these objects will be resident in the main-memory buffer.

- The flexibility in chunk size for large objects (subobject size in Vagabond) makes management of large multidimensional arrays (including OLAP data cubes) efficient.

- Compression can be applied to sparse data, for example objects where many attributes have a NULL value.

16.1.4 Web/XML

An ODBMS is well suited for storage of tree/DAG structures such as Web/XML documents. In addition, Vagabond has features that can make Web/XML data management more efficient than in other ODBMSs:

- Much of the data stored as XML is actually text data, and when querying such data efficient text search is important. Storing signatures in the OIDX can significantly reduce the cost of such queries.
- As pointed out by Abiteboul [1], we are often more concerned with querying the recent changes in some data source than in examining the entire source. Support for temporal data in the storage layer facilitates this.

16.1.5 Summary

Based on the analysis above, we conclude that the Vagabond approach is a very suitable solution for the described application areas. How well it will perform in traditional application areas with high update rates and no need for temporal support will depend very much on the amount of buffer memory available. To be competitive in applications with non-temporal data, most of the objects *and* the OIDX have to be resident in main memory. If this is not the case, an ODBMS based on the in-place update approach is likely to provide a higher performance.

16.2 Contributions and Publications

The contributions of this thesis can be summarized as follows:

- Log-only and write-optimized ODBMSs, and the design of Vagabond, also presented at DEXA'97 [158], SCCC'97 [159], NIK'99 [157], and HiPOD'2000 [149].
- A comparison of the performance of log-only and in-place update based ODBMSs, also presented at SSDBM'2000 [153] and CIKM'2000 [150].
- OID indexing in temporal ODBMSs, also to be presented at IDEAS'2000 [155].
- Persistent caching of index entries, also presented at VLDB'99 [148].
- Object declustering in parallel temporal ODBMSs.
- Transaction processing in log-only ODBMSs.
- Aggregate and grouping functions in ODBMSs, also presented at SCCC'96 [163].
- Techniques for improving and optimizing partitioning in database query processing, presented at NIK'96 [31] and BNCOD'97 [30].[1]
- Optimizing OID index access performance in page server based ODBMSs, also presented at DEXA'98 [160].

[1]The major contributions in the NIK'96 and BNCOD'97 papers come from the first author, Prof. Kjell Bratbergsengen.

- OID index entry caching in temporal ODBMSs, also presented at FODO'98 [161] (a revised version of the FODO'98 paper is also published in [162]) and at ADBIS-DASFAA'2000 [152].
- The use of signatures in the OID index, also presented at ADBIS'99 [147].
- Identification of issues in temporal object management that should be further investigated in order to increase the performance of a temporal ODBMS, including clustering of temporal objects and language bindings, also presented at ADBIS-DASFAA'2000 [151].

During the work with this thesis, we have also done other related work, which is not included in this thesis. This includes a study of concurrency control strategies in distributed ODBMSs, presented at ADBIS'97 [164], and a description and analysis of the signature cache approach [154, 156].

16.3 Criticism

The main criticism against this thesis is that the proposed system has not yet been implemented, and that the analytical models have only in part been validated by simulations. An implementation would serve as a better proof than analytical modeling of the advantages and performance gain from the proposed approach. In addition, it would help in answering other problems that we have not had the time to analyze thoroughly:

- Is the system described here is too complex, making it CPU bound? This argument is more against the flexibility provided by the proposed system than the log-only approach itself. It should also be noted that there are room for improvement in the management of the physical data structures. The description in this thesis has not focused on management and "fine tuning" of the structures.
- The cleaner process is run in the background, and employs idle resources. However, under certain conditions, its cost can be significant.
- Maybe the "everything is an object" philosophy has been taken too far? In some cases, using special structures for indexes instead of using objects could improve performance as well as making implementation both cleaner and easier.
- It is difficult to know what kind of access patterns that will be experienced in future temporal ODBMSs. It is possible to do predictions based on current access patterns, but we believe that it is quite possible that when support for temporal features become common, application developers will utilize these in new ways. We do not claim that the solutions in this thesis are optimal in terms of performance, and in particular, we expect that it is possible to improve the OIDX when we know more about typical access patterns.
- The proposed Vagabond Temporal OID Index (VTOIDX) has not been analyzed or compared against other indexes, for example the TSB-tree [132].
- Lock durations and high concurrency aspects in the context of the OIDX.
- We have also experienced one of the possible disadvantages of using a log-only approach: it is more difficult to separate the logical modules, for example buffer management, from each other in such a system. This is an observation that has also been done in LFS related research [183].

16.4 Future Work

In many ways, we have a feeling that the work described in this thesis has raised more questions than have been answered. There is no doubt that there is work for many doctoral theses in this area. In addition to the topics listed in the previous section, some ideas that we would like to explore further are:

- Should the OIDX be log-only or based on in-place updating?
- Will the OIDX be a bottleneck even when an OD Cache, a PCache, and the VTOIDX are used?
- Using Vagabond as a valid time or bitemporal ODBMS.
- How to do prefetching in an efficient way.
- Versioning of indexes, and the semantics of this.
- Using the index entry caching techniques in this thesis in the context of indexing in a traditional DBMS.
- Language bindings for temporal data access.

Vagabond is designed as a parallel and distributed ODBMS, and that raises a lot of interesting issued that should be studied further. Important examples are:

- Caching of ODs on remote nodes. This can be interesting for many reasons, but in our context, a new reason is that the caching can be used to benefit from the signatures stored in the ODs.
- Allow objects and object versions to be written anywhere, and maybe no home node at all? Here load balancing is an important issue.
- Partitioning of large objects. Maybe use separate OIDs of subobjects, so that large objects can be realized as multiple subobjects internally. This could make replication and indexing easier.
- Approaches for declustering the log instead of its contents.

Part IV

Appendixes and Additional Papers

Appendix A

Aggregate and Grouping Functions in Object-Oriented Databases

This appendix contains the paper presented at the XVI International Conference of the Chilean Computer Science Society (SCCC'96), held in Valdivia, Chile, November 1996.

Aggregate and Grouping Functions in Object-Oriented Databases

Kjetil Nørvåg and Kjell Bratbergsengen
Norwegian University of Science and Technology
Department of Computer Science
N-7034 Trondheim, Norway
{noervaag,kjellb}@idt.ntnu.no

Abstract

Efficient evaluation of aggregate functions in object-oriented databases (OODB) can have considerable impact on performance in many application areas, like geographic information systems and statistical and scientific databases. The problem with current systems is inefficient execution of aggregate functions with large data volumes, and lack of flexibility: it is not possible to extend the systems with new aggregate and grouping functions. In this paper, we extend the concept of aggregate functions from relational databases. We introduce the concept of grouping functions, which could enhance flexibility and performance considerably. We show how this could be implemented into an OODB. We also describe how support for special kinds of aggregate queries and data structures can help in designing future high-performance systems.

Keywords and phrases: aggregate functions, object-oriented databases, database programming languages, query processing

1 Introduction

Object-oriented database systems (OODB) are attractive alternatives to traditional relational database systems. This is especially true for applications where the modeling power of relational databases is insufficient, and the language mismatch makes integration between the application programs and the database system difficult.

OODB have been an immature technology, and have only recently been put into "real work". But: in some application areas relational databases still beat them in performance. Common for many of these application areas are heavy use of *aggregate functions*, together with *grouping*. A typical aggregation query partitions a collection of objects, and evaluates one or more functions on the objects of a group. Typical examples are sum or average of an attribute in each object in a group. The result is a new collection of objects, one for each group. Although in most database systems these are not the most commonly executed operations, queries with aggregate functions possibly scan large parts of the database, making them very time consuming.

Traditionally, business applications have have been the most heavy users of aggregate functions. In the last years, new application areas have emerged. Areas of particular interest are geographic information systems, data mining, data warehousing, and statistical and scientific databases[1]. These databases contain highly structured data, very suitable for the object-oriented data model. Operations on (and analysis of) data in these systems make heavy use of mathematical and statistical operators, and in some applications, complex grouping. To be able to do efficient queries, these operators should be a part of the query system.

As of today, most OODB provide support for execution of basic aggregate functions, but often with low performance when applied to large data volumes. Also, the systems lack the desired flexibility: it should be possible to extend the systems with new aggregate and grouping functions.

[1] It should be noted that the examples described here are not necessarily independent, if you have a data warehouse, it is very likely you want to do analysis on it, by the use of data mining techniques.

In this paper, we extend the concept of aggregate functions from relational databases, and we introduce the concept of *grouping functions*. The use of grouping functions can enhance flexibility and performance considerably. We show how this could be implemented into an OODB. Finally, we describe how support for special kinds of aggregate queries and data structures can help in designing future high-performance systems.

2 Aggregate Functions in OODB Research

Much research has been done in aggregate function evaluation in relational databases, but evaluation of aggregate functions in OODB is still an immature area of research. The main reasons are probably:

1. *The importance of efficient evaluation of aggregate functions in OODB has not been recognized.* Application areas suitable for OODB, like those discussed in the previous section, should justify the importance.

2. *The advantages of storing and processing data in databases instead of files has not been recognized in all applications areas where it is appropriate [5].* The focus in scientific computing has been on doing computations with files as inputs. Especially with complex file formats as, e.g., the HDF file format[2], computations and maintenance is not trivial. It is also worth noting that important projects, as e.g. EOSDIS[3], are drifting from files to database storage of data.

3. *It is often thought that the algorithms developed for relational databases are good enough for OODB as well, if the aggregation is done on primitive attributes.* If, on the other hand, the aggregation is done on methods, the problem is thought to be equivalent in complexity to a general read-only query. One should keep in mind that the evaluation of aggregate functions form a restricted subclass of read-only queries, which gives room for considerable improvement.

Although the effort put into research on aggregate functions in OODB has been low, one should keep in mind that much has been published about topics related to aggregation. The most important work is, of course, research done on aggregate function evaluation in relational databases. The fundamentals are the same, and many results valid for relational databases applies to OODB as well. Valuable sources for aggregate function evaluation are Bitton et.al. [1], Bratbergsengen [3] and Shatdal et.al. [16]. Only recently has the grouping functions been the focus for query optimization [23, 9, 10].

3 Aggregate and Grouping Functions

Aggregation is basically partitioning a set of objects into groups, and evaluate one or several aggregate functions over the objects in the groups. The result is one set of values (from the aggregate functions) for each group.

Aggregate Functions In our notation, we will write aggregate functions as $\mathtt{A}(\mathtt{P}^A)$. For each function $\mathtt{A}$, a set of *parameters* is given from the set $\mathtt{P}^A$, which are one or more attributes from the objects that the aggregate function is applied to.

All system with query languages already support some aggregate functions, like sum and average, but with new application areas, with very different needs, it is important to have the possibility

[2] HDF is a file format for storing and transmitting scientific data sets, and a library interface for working with the data [5].

[3] Earth Observing System Data Information System

of adding new functions. Examples are statistical functions in statistical and scientific databases, spatial functions as area and perimeter in GIS/spatial databases, and special time-related functions in temporal databases.

It is useful to think of aggregate functions as state transition functions, which in e.g. POSTGRES [24] is done explicit by the use of the CREATE AGGREGATE construction. The difference between an aggregate function and an ordinary function is that an *initial state* has to exist. With the concept of objects available, we can define an aggregate function as an object class, here called *aggregate function objects*. A new aggregate function object, with its defined initial state, is created for each group during execution. An aggregate function class should contain:

1. A *constructor* that defines an initial state.
2. An *iterator* method to be called for each object aggregated to the group.
3. A *method returning the final result* of the aggregation for the group.
4. With two of the actual parallel aggregation algorithms, more than one aggregate object can exist for each group (we will come back to this in section 6.4). As a part of these algorithms, it must be possible to merge these preliminary group objects into one. To be able to do this, with general aggregate functions, it is necessary to have *merge methods* defined, as well.

Grouping Functions Which group an object belongs to, is determined by the result of one or more grouping function $\mathtt{G}(\mathtt{P}^G)$. Each function `G` has parameters from the parameter set $\mathtt{P}^G$. P^G consists of one or more attributes from the objects which aggregation is applied to, together with (optional) parameters to the grouping function. The latter ones are independent of the values of the objects, they are constant values during the aggregation. Grouping functions have no states, they are just mapping functions from a possible multi-dimensional space with continuous-valued attributes to a space with discrete-valued attributes, returning a group identifying result.

In current database systems, there is really no such thing as grouping functions. Which group an object should belong to, is determined by a concatenation of the value of one or more attributes. In the context of grouping functions, we can say that this is the identity grouping function, `I(p) = p`, where output equals input. In the future, it should be possible to define grouping functions in the same way as aggregate functions.

In applications where several combinations of attributes should map into the same group, grouping functions are necessary to avoid a costly preprocessing of data. Example applications are spatial and temporal databases. If we want coordinates within an area to belong to the same group, we can have a grouping function that do this mapping. Another example is temporal grouping, where all data in a fixed interval of time should map to the same group. The described grouping functions can also be extended to grouping where one object can participate in more than one group. This can be useful for several application areas, e.g. some business data warehousing applications.

One important aspect of grouping functions that returns a group identifier, is that this kind of grouping lend itself easy to hash-based algorithms. This is important to get high performance.

4 Query Processing Operator

It is useful to explain (and implement) aggregation and other operations as language-independent *operators*. This is also the way it is done in systems as Gamma [6] and Volcano [8]. A stream of objects flows through a tree of operators. An "object" can be either a materialized object, or the object identifier. In OODBs it should be is possible to operate on methods, in the same way as on attributes. When we in the rest of the paper talk about attributes, these could be either values, or method invocations.

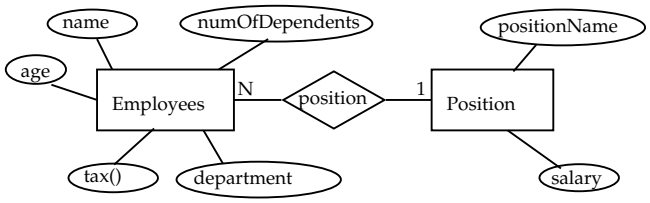


Figure 1: Example database

The aggregate operator[4] can be written as:

$$\texttt{AGGREGATE(I:A}_1\texttt{(P}_1^A\texttt{),...,A}_n\texttt{(P}_n^A\texttt{):G}_1\texttt{(P}_1^G\texttt{),...,G}_m\texttt{(P}_m^G\texttt{):R)}$$

where:

- `I` is the input stream. This can be a stream from another operator, e.g. `SELECT`.
- `A` defines a set of aggregate functions. For each function, a set of *parameters* is given from the set $\texttt{P}_i^A$.
- `G` is the set of grouping functions. The grouping functions are applied to all objects in the input stream, and the set of results determines for each object which group it belongs to: $\mathrm{G} = \cup_{i=0}^{m} G_i$. The concatenation of partial results from the grouping functions should probably be mapped to a group identifier. This makes hash-based aggregation algorithms easier to employ.
- `R` is the output stream. The aggregate functions `A` are applied on every set of objects belonging to the same group, thus giving one result object for each group. Each object contains `G` and the result of `A` applied on the set. Thus, the schema is $\mathrm{R}_s(\mathrm{G}, \mathrm{A}_1, \ldots, \mathrm{A}_n)$.

5 Aggregation in Query and Application Languages

Relational databases usually have support for aggregate functions in their query language (SQL), although efficiency differ. Most object-oriented databases have (or will soon have) support for aggregate functions with grouping in their query language, but lacks support in their application languages, requiring the user to explicitly call the query language from the application language, giving an impedance mismatch. In the ODMG-93 Object Query Language [4] the operations are supported by a group-by-expression similar to SQL.

To illustrate notation and languages, we will use the employee database in Figure 1 as an example. This database has two classes: The class `Employee`, with attributes `name`, `department`, `age` and `numOfDependents`, and the class `Position` with attributes `positionName` and `salary`. In addition, we have a relationship between the lasses as depicted by the figure. The method named `tax()` in the `Employee` class, returns the tax as a function of `numOfDependents` and the salary in the `Position` object. We call the extent (collection of all `Employee` objects) `Employees`. Suppose we wanted a list of the average age of people in the different departments in our example database. With the notation from Section 4, and the result put into `Result`, this can be written:

```
AGGREGATE(Employees:avg(age):I(department):Result)
```

[4]This operator is based on the aggregate operator described by Bratbergsengen in [3].

Query Language In the ODMG-93 Object Query language (Release 1.2), this can be done with the following query:

```
select department, avg_age: avg(select age from partition)
from Employees e
group by dep: e.department
```

Application Language It is important to have the aggregate operator as a part of the application language to avoid the impedance mismatch. As of today, the user is usually required to write the code himself or get the result through a call to the query processor. In that case, the query is formulated as a string interpreted at run-time. This is the case of the current version of the ODMG C++ binding [4]. If we embed the previous query into a C++-program, with the result of the query is returned in `resultSet`, we could write this as:

```
Set<Ref<DepAvgAge>> resultSet;
oql(resultSet, "select department, avg_age: avg(select age from partition)\
                from Employees e\
                group by dep: e.department");
```

Use of Methods in Queries In an OODB, it is also possible to do queries on methods, where methods should be possible to use just the same way as attributes. If we wanted a a list of the average amount of tax paid by the people in the different departments, it should be possible to do this as:

```
select department, avg_tax: avg(select tax() from partition)
from Employees e
group by dep: e.department
```

6 Execution of Aggregation Queries

In the relational data model, all tuples streaming to an aggregate operator contain atomic attributes, which can be processed immediately, and without accessing other tuples. This is not the case in the object-oriented data model. If the attributes are just values, this corresponds to retrieval of tuple attributes, but evaluation of methods is more difficult. The methods might involve computations, and possibly access to other objects. To make the aggregation part "cleaner", we separate the evaluation of methods and the aggregation by introducing a *materialize* operator. This operator outputs a restricted materialization of the objects: the required methods, and no other, are evaluated, and the results are output to the aggregate operator. In this way, the aggregate operator can apply the aggregation stream without computations and without accessing other objects. The result is that we can use a aggregate operator similar to the relational one.

6.1 The Materialize Operator

The straightforward strategy is to naively fetch an object, evaluate the methods, output the result, and continue with the next object in the stream. As long as the methods does not access other objects, this strategy works well. But if methods accesses other objects (which is similar to a functional join), this might be inefficient: if an object is accessed from more than one object in the aggregate stream, it might have been thrown out from the buffer/cache between the accesses. The object has to be re-read from disk, which is inefficient. Therefore, we want to control these accesses. This can be done by employing available techniques for functional join. Several algorithms and techniques are described by Shekita et.al. in [19, 18, 17], and Lieuwen et. al. in [14, 13].

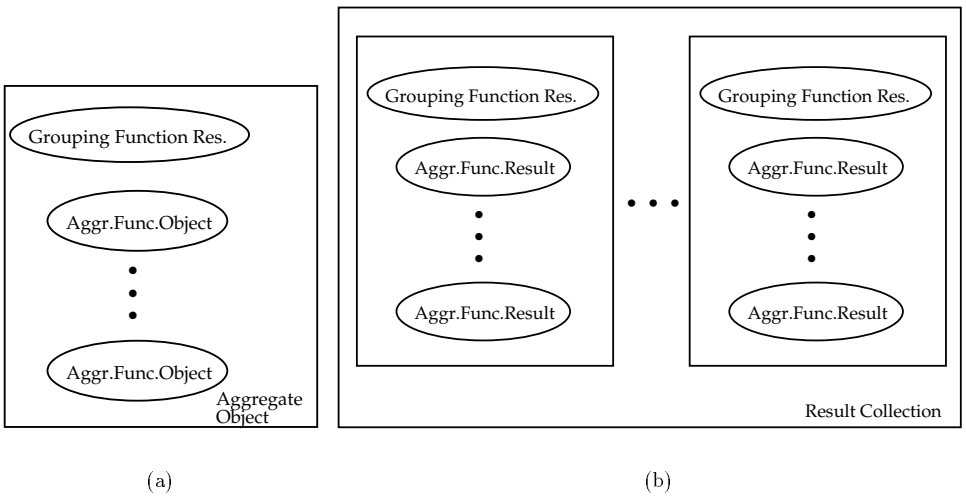


Figure 2: Aggregate object and objects in result collection

6.2 The Structure of Aggregate and Result Objects

During aggregation, a new *aggregate object* is created for each group (see Figure 2a). The aggregate objects will belong to a class with the following attributes:

1. An attribute identifying the group (the result of the grouping functions applied on an object).
2. One attribute for each A_i in the list of aggregate functions. These attributes are object instances of aggregate function classes.

The result of the aggregation is a collection with one element for each group (see Figure 2b). These objects contains the final results of the aggregate functions. The result objects will belong to a class with the following attributes:

1. An attribute identifying the group .
2. One attribute for each A_i in the list of aggregate functions. These are the final results of the aggregate functions.

6.3 The Aggregate Operator

The function of the aggregate operator is to apply the aggregate function(s) on the objects it receives, and group according to the grouping function(s). In the relational data model, all tuples streaming to an aggregate operator contain atomic attributes. With the use of the materialize operator, aggregation in OODB can be done with the traditional algorithms used in relational databases.

The *nested-loop aggregation algorithm* is the basic approach, but becomes inefficient when the size of the aggregate objects (number of aggregate objects$*$size of each aggregate object) is much larger than primary memory. In that case, one should either

- *pre-sort* the objects in the object stream based on the grouping identifier, or
- *partition* the objects in the input stream based on the grouping identifier

before aggregation. If the input stream is not sorted initially, which is common in an OODB[5], it is better to use a partition-aggregate algorithm. With un-sorted input, partition-aggregate will always perform better than sort-aggregate [2].

6.4 Parallel Aggregation

In general, it is difficult to parallelize navigational queries in OODB [11]. Set processing, on the other hand, lends itself to parallelizing. With parallel machines, distribution of work between nodes can improve performance. Several algorithms for distributed aggregation exist, with the following in common use:

- Parallel aggregation with local aggregation/central coordinator
- Parallel aggregation by redistribution
- Redistribution with local aggregation

With parallel aggregation in OODB, we get a problem that does not exist in relational databases: how to evaluate methods in objects accessed from objects to be aggregated. There are several ways to evaluate these methods, we have here the query vs. data shipping problem. This has much in common with distributed (functional) join, but the the use of complex methods complicates the situation.

Local Aggregation/Central Coordinator This algorithm is only useful for scalar aggregates, e.g., with no grouping. All nodes do local aggregation on the part of the data allocated to that node. The local aggregation can be done with one of the three algorithms described above. The resulting aggregate objects are sent to a coordinator node, which uses the preliminary aggregate objects it receives to create the final aggregate objects. With user defined aggregate functions, it is necessary to have available methods to merge the partial preliminary objects (see section 4).

Redistribution In the first phase, objects are relocated to nodes according to a hashing function applied on the result of the grouping methods. Objects belonging to a group is, in that way, guaranteed to end up at the same node. In the second phase, aggregation is performed at each node with one of the three first algorithms.

Redistribution with Local Aggregation In the parallel aggregation by redistribution algorithm, more data than necessary is moved. With large groups (many objects in each group), we can gain much by doing local aggregation before redistribution [3]. This is the way aggregation with grouping is done in e.g. Gamma [6]. In the first phase, all nodes do local aggregation on data residing on the node, just as in the local aggregation/central coordinator algorithm. In the second phase, the result is distributed according to a hashing function applied on the value of the result of the grouping methods. The global result is obtained by merging the preliminary objects (cf. section 4) as done in parallel aggregation by redistribution.

Several implementations have shown that with high-bandwidth communication between nodes, the local aggregation is not necessarily beneficial. Whether to do local aggregation or not depends on data selectivity and communication bandwidth. As suggested by [16], sampling of data should be used to decide if local aggregation should be done before redistribution.

[5]Sort-aggregate is the algorithm used in most existing commercial relational systems [7]. One reason for this, is that quite often, traditional applications want a sorted result from the aggregation (ORDER BY in SQL). In a typical OODB application, this will probably *not* be the case.

7 High-Performance Aggregation

The performance of evaluation of aggregate functions can be improved in several ways. We have already treated parallel aggregation, in addition it is useful to study optimization for special kinds of aggregate queries, special access patterns, and special data structures:

Multi-Level Aggregation Sometimes, aggregation is nested, *multi-level aggregation*. That is, aggregation done on a group of aggregate groups. This can be exploited to increase performance.

Aggregation on Multi-Dimensional Data Structures Although current databases only support aggregation on collections as sets or bags, the aggregation can be done on other multi-dimensional structures. This is useful for spatial and temporal databases.

In OODB, it is possible for the user to specify aggregate functions. While this works well on set-like structures, it is likely to give bad performance on other structures. The best alternative might be to provide new aggregate operators that work on non-set-like structures.

Temporal Aggregates Conventional aggregate algorithms are not efficient when applied to temporal databases. Algorithms for computing temporal aggregates are presented by Kline and Snodgrass in [12]. As far as we know, no one has published work on parallel algorithms for temporal aggregation, or algorithms for temporal aggregation in OODBs. Temporal databases can be viewed as a subclass of multidimensional databases, and algorithms similar to aggregation on spatial data structures can be employed.

Combining Bulk Loading and Aggregation Bulk loading is *loading a large external dataset during a single sitting* [15]. If it is known at load time what kind of aggregate functions that will be employed later on the data set, it is possible to do aggregation while loading. Combining bulk loading algorithms with aggregation will probably be beneficial, especially in databases where summary data can be exploited (e.g. in statistical and scientific databases). As a starting point, the bulk loading algorithms developed by J. Wiener [22] can be used. But, no good algorithms for parallel loading has yet been developed. This problem has to be solved, as a single-threaded algorithm will be a potential performance bottleneck.

Precomputed Results In many of the proposed application areas for aggregate evaluation, much of the data will be static. It is possible to take advantage of this by either having precomputed the query for part of the database, and/or use stored results from earlier queries. These techniques are quite similar to techniques used for view maintenance/materialized views in data warehousing [21, 9, 10].

Another approach is to have the system maintain an index to the data with statistical summary data.[6] By using a structure which makes it easy to exploit the summary data (a tree is used in the tree based access method proposed by Srivastava and Lum [20]), performance can be greatly enhanced if the data are heavily used.

Resumable Aggregation In large databases, queries involving aggregation can be a very time consuming. It is desirable that a crash during aggregation does not mean that all the work is wasted. Several approaches to avoid this can be used. Examples are aggregation checkpointing and storing partial results. The partial results is similar to materialized views, but for resumable aggregation they are completely machine generated.

[6] Statistical summary data is partial results needed for some operations. An example is a sum of some elements.

8 Conclusions

Efficient evaluation of aggregate functions in object-oriented databases (OODB) can have considerable impact on performance in many application areas. Still, current systems are not able to compete in performance with traditional relation database systems. In this paper, we have justified the need to concentrate on research in this area, and also showed why so little previous research exists.

In this paper we have extended the concept of aggregate functions from relational databases, and introduced the concept of *grouping functions*. Grouping functions are not currently offered by any system, but in this paper we have shown how this could be integrated into the aggregation process. By offering aggregation and grouping as depicted in this paper, the systems will be able to provide the desired flexibility needed in future high-performance database applications. We believe, that the way to high performance aggregation in future database systems lies in:

- More efficient algorithms. Aggregation on methods has much in common with functional join, and often functional join is part of the process too. It will be beneficial to use resources on this, and in particular on parallel and distributed functional join.
- Maintaining and employing precomputed results. When applicable, this is probably the strategy that can give most in increased performance. The implementation cost need not be too high. These precomputed results has to be automatically created when necessary by the system.
- Developing new algorithms for applications with multi-dimensional data structures. These algorithms will also be useful for evaluating temporal aggregates.
- More research in grouping functions. Grouping functions, with mapping to a discrete domain, lend itself easy to hash-based aggregation algorithms.

References

[1] D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson. Parallel Algorithms for the Execution of Relational Database Operations. *ACM Transactions on Database Systems*, 8(3), 1983.

[2] K. Bratbergsengen. Hashing Methods and Relational Algebra Operations. In *Proceedings of the 10th International Conference on VLDB*, 1984.

[3] K. Bratbergsengen. Relational Algebra Operations. In *PRISMA Project Workshop, Nordwijk, The Netherlands*, 1990.

[4] R. Cattell, editor. *The Object Database Standard: ODMG-93. Release 1.2.* Morgan Kaufmann, 1996.

[5] D. J. DeWitt. DBMS - Roadkill on the Information Superhighway. Invited talk at VLDB'95, 1995.

[6] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[7] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), 1993.

[8] G. Graefe. Volcano — An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1), 1994.

[9] A. Gupta, V. Harinarayan, and D. Quass. Generalized Projections: A Powerful Approach To Aggregation. Technical report, Stanford, 1994.

[10] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-Query Processing in Data Warehousing Environments. In *Proceedings of the 21st International Conference on Very Large Data Bases*, 1995.

[11] K.-C. Kim. Parallelism in Object-Oriented Query Processing. In *IEEE Sixth International Conference on Data Engineering*, 1990.

[12] N. Kline and R. T. Snodgrass. Computing Temporal Aggregates. In *Proceedings of IEEE 11th Int'l Conference on Data Engineering, March, 1995*, 1995.

[13] D. Lieuwen, D. DeWitt, and M. Mehta. Pointer-based Join Techniques for Object-Oriented Databases. Technical Report CS-TR-92-1099, University of Wisconsin-Madison, 1992.

[14] D. Lieuwen, D. DeWitt, and M. Mehta. Parallel Pointer-based Join Techniques for Object-Oriented Databases. In *Proc. 2nd International Conference on Parallel and Distributed Information Systems*, 1993.

[15] D. Maier and D. M. Hansen. Bambi Meets Godzilla: Object Databases for Scientific Computing. In *Seventh International Working Conference on Scientific and Statistical Database Management*. IEEE Computer Society Press, 1994.

[16] A. Shatdal and J. F. Naughton. Adaptive Parallel Aggregation Algorithms. In *Proceedings of the 1995 ACM SIGMOD*, pages 104–114. ACM Press, 1995.

[17] E. Shekita. *High-Performance Implementation Techniques for Next-Generation Database Systems*. PhD thesis, University of Wisconsin-Madison, 1991.

[18] E. J. Shekita and M. J. Carey. Performance Enhancement Through Replication in an Object-Oriented DBMS. In *Proceedings of the 1989 ACM SIGMOD*, 1989.

[19] E. J. Shekita and M. J. Carey. A Performance Evaluation of Pointer-Based Joins. Technical Report 916, University of Wisconsin-Madison, 1990.

[20] J. Srivastava and V. Y. Lum. A Tree Based Access Method (TBSAM) for Fast Processing of Aggregate Queries. In *IEEE Fourth International Conference on Data Engineering*, 1988.

[21] J. Widom. Research Problems in Data Warehousing. In *Proceedings of the 4th Int'l Conference on Information and Knowledge Management (CIKM), November 1995*, 1995.

[22] J. L. Wiener. *Algorithms for Loading Object Databases*. PhD thesis, University of Wisconsin-Madison, 1995.

[23] W. Yan and P.-Å. Larson. Eager Aggregation and Lazy Aggregation. In *Proceedings of the 21st International Conference on Very Large Data Bases*, 1995.

[24] A. Yu and J. Chen. *The POSTGRES95 User Manual*, 1995.

Appendix B

Improved and Optimized Partitioning Techniques in Database Query Processing

This appendix contains the paper presented at the Fifteenth British National Conference on Databases (BNCOD15), held in London, July 1997.

Improved and Optimized Partitioning Techniques in Database Query Processing

Kjell Bratbergsengen and Kjetil Nørvåg

Department of Computer Science
Norwegian University of Science and Technology,
7034 Trondheim, Norway
{kjellb,noervaag}@idt.unit.no

Abstract. In this paper we present two improvements to the partitioning process: 1) A new dynamic buffer management strategy is employed to increase the average block size of I/O-transfers to temporary files, and 2) An optimal switching between three different variants of the partitioning methods that ensures minimal partitioning cost. The expected performance gain resulting from the new management strategy is about 30% for a reasonable resource configuration. The performance gain decreases with increasing available buffer space. The different partitioning strategies (partial partitioning or hybrid hashing, one pass partitioning, and multipass partitioning) are analyzed, and we present the optimal working range for these, as a function of operand volume and available memory.

Keywords: Relational algebra, partitioning methods, buffer management, query processing

1 Introduction

Relational algebra operations are time and resource consuming, especially when done on large operand volumes. In this paper, we present a new, dynamic, buffer management strategy for partitioning which can significantly reduce the execution time: *the circular track snow plow strategy.* Our approach is motivated from three observations:

1. When doing disk intensive operations, it is advantageous to process as large blocks as possible when doing disk accesses. Our strategy uses an optimal partitioning, which gives as large blocks as possible.
2. With the traditional fixed size block methods, only half the available memory is actually holding records. Our strategy, with variable size blocks, will with the same amount of available memory, double the average block size, and thus make much better use of available memory resources.
3. While it might be true that main memory on computers are large, and getting even larger, not all of this is available for one relational algebra operation. Often, several programs are running concurrently, several users are running queries concurrently, and queries can be quite complex, resulting in a large

tree of query operators. In this case, available memory for each operator can be rather small. As the comparison between the methods will show later in the paper, our method will be especially advantageous with small amounts of memory available, but it will always perform better than the traditional approach.

In the rest of the paper, we first present some related work in Sect. 2, and give an introduction to partitioning strategies in Sect. 3. The circular track snow plow strategy is introduced in Sect. 4. We present our cost model and assumptions in Sect. 5, and derive cost functions for the partitioning strategies. Finally, we compare these approaches in Sect. 6, and show a significant performance gain by using the snow plow strategy.

2 Related Work

Optimal splitting of source relations is discussed by Nakayama et. al. in [9]. Their conclusion is to partition the relation into as many partitions as possible. They say nothing about the block size, except that it is fixed. Their algorithms is beneficial to use when we have heavy data skew, but in other situations the small block size makes them expensive, as pointed out by Graefe [5]. Block size (clusters) has usually been determined from experiments, simulations, or just common sense, with Volcano [6] as an exception. Recently, the use of variable block size has also been recognized and studied by Davidson and Graefe [3], and Zeller and Gray [11], but these papers focuses on memory availability, rather than a thorough analysis by the use of cost functions.

3 Partitioning Strategies

With smaller operand volumes, nested loop methods are superior, requiring only one scan of the operands to create the resulting table. When the operand volume increases, nested loop methods are still employed in the final stage, but now after a hash partitioning stage, as described in [2, 1, 4].

Several partitioning strategies exists. They can be classified as *no splitting*, *partial, one pass, and multipass partitioning*. This first one, no splitting, is used when the smallest operand is less than or equal to available workspace. When this is the case, the whole operation can be done in main memory.

When the smallest operand is larger than available memory, one can split in one pass. All available workspace is used for splitting. We call this variant *one pass partitioning*. The number of partitions, p, is so large that each partition can be held in work space in the next stage.

When the smallest operand V is larger than the memory M, but less than some limit V_{ppu}, partial splitting (or hybrid hash) can be employed. Part of the memory is used for partitioning, the other part is used for performing the relational algebra operation. When the operand gets larger, more work space

area is needed for splitting. The upper limit V_{ppu} is found when all available memory is best used for splitting the operand.

As the smallest operand gets very large, a large number of partitions is necessary when one pass partitioning is employed. The result is a small block size, because the block size $b = M/p$ (or, as we will see later, with our method, $b = 2M/p$). As this block size gets smaller, there is a limit where splitting is best done in several passes, *multipass partitioning*.

It is useful to classify the (smallest) operand size, relative to available workspace M, in four classes: small, medium, large, and huge operands. As is shown in Table 3, partitioning strategy is determined from operand class. How to decide the bounds for each class will be shown later in the paper. As pointed out in the introduction, it is important to keep in mind that not all of the main memory is available as workspace for the partitioning process.

Operand Class	Size of Smallest Operand	Strategy
Small	$V \leq M$	No partitioning
Medium	$M < V \leq V_{ppu}$	Partial partitioning
Large	$V_{ppu} < V \leq V_{opu}$	One pass partitioning
Huge	$V_{opu} < V$	Multipass partitioning

Table 1. Operand classes and corresponding partitioning strategies.

4 The Circular Track Snow Plow Strategy

With the traditional splitting strategy, as described in [2, 1, 7, 4, 10], we have a fixed size of memory for each group. When a new tuple arrive, it is moved to its block buffer. Whenever a block buffer is full, it is written to disk.

If we look closer at the fixed block size splitting, we see that only about half the available memory is actually holding records. If we do not divide the memory into fixed sized block buffers, but rather let work space be one common memory pool, we do not have to write records to disks until all memory is taken. Then we write the first group to disk. We now have a new period where all groups (part of partitions in memory) are growing at approximately the same speed. This holds if the hash partitioning formula gives an even distribution. Again when there is no room left, the next group is written to disk. After all groups have been written once to disk, we start over again with the first group. After a transient start up phase, we can see that the average size of the groups written to disk is approximately $b = 2M/p$. This method is analogous to the replacement selection sort which is used for initial sorting in sort-merge programs. Why the average block size is $2M/p$, is well described by Knuth in [8]. The situation can be compared to a snow plow on a circular track of length l. The plow is always plowing full depth snow. Just behind the plow, the depth is zero, and the average snow depth on the track is one half the full depth h. The total amount of snow on the track is $hl/2$. In our case, the full snow depth h corresponds to the block

size b, and the track length l corresponds to the number of subfiles p. Then $bp/2 = M \Rightarrow b = 2M/p$. This is illustrated in Fig. 1, which shows the groups in memory during splitting. To the left we see the situation just before the first group (0) is written to disk. To the right we see a steady state situation just after group 3 has been written to disk. The next group to go is number 4, when the lower limit of available space is reached.

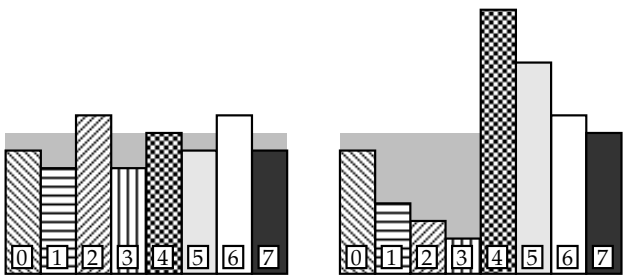


Fig. 1. Groups in memory during splitting.

4.1 Memory management

The memory is now used as a heap, storing records. The records are logically separated into p groups. This can be done using linked lists, pointer arrays, linked lists with pointers, etc. The group is determined using a hash formula on the operand key. Records are read and stored in memory until the amount of free memory reaches a lower limit. In the stationary situation, the amount of memory used is the same as the amount released. The addresses of the free space slots could be stored on a small stack. Memory management is especially simple if all records have the same size, however, if there are variable length records, more elaborate schemes should be used. Memory management at this level is important, because it could take a lions share of CPU time.

4.2 Writing Blocks to Disk

To take full advantage of this method, each group should be written to the disk as one block. Because of the stochastic nature of the group size, groups do not in general have the same size. Also, in the initialization phase, the average group size is rapidly growing from M/p to $2M/p$.

Despite the groups in memory are of different sizes, each group could be stored as a chain of fixed sized blocks on the disk. We should set aside two buffers to allow for double buffering. Each buffer should be $2M/p$ bytes. When a group is ready for output, its records are moved to the free output buffer. In the startup transient phase, we would not be able to fill the buffer before we

have to write it, but in the stationary phase most of the time it will be nearly full. Sometimes we have to leave some records of the group behind, they will have to go to the next block for that group. To better fill the output buffer, we could change the *round robin* sequence strategy for writing groups to disk, to a *largest group first strategy* (LGF). This strategy might even lead to a larger average block size than $2M/p$.

It would be even better if we could get user level functionality enabling us directly to write to, and read from, the SCSI port. This requires some redesign of the ASPI interface. The current ASPI interface provides only a traditional block transfer I/O command. The command specifies a block address, buffer address and block length. No constructive interaction is possible until the command has finished, i.e. transferred a complete block. We would need new functionality, like moving single words or smaller blocks between SCSI port and user memory. This is effectively a *gather write* function available at the user level. This could save a lot of unnecessary copying.

4.3 Disk Layout

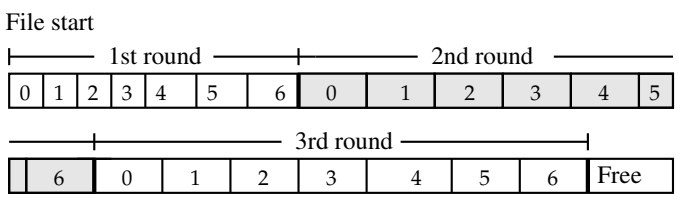


Fig. 2. The disk image after all groups have been emptied three times.

If we write the disk with fixed sized blocks, there is no problem finding all blocks of one group, they are chained together. The following is about disk layout when the disk is written with variable sized blocks (groups). When the groups are written to disk, we will get a pattern similar to that shown in Fig. 2, if neighboring groups are written contiguously onto disk. A *Round* in the figure is *one round around the circular track*, processing each group once. The numbers are group numbers. Writing groups contiguously gives a bonus when we flush all groups at end of input. This can be done in one operation. It is also necessary if we want to create *supergroups*, by joining neighboring groups into one supergroup. This is useful when the operand(s) has been over-partitioned (the operand has been partitioned into a larger number of partitions than necessary). This can happen if we are uncertain about the number of subfiles we need, and choose to overpartition to err on the right side. The overpartitioning causes more disk accesses than necessary during the splitting phase. The extra cost of overpartitioning should be removed from the reading phase, and we can

Round	Address	g0	g1	g2	g3	g4	g5	g6
0	0	10	12	14	15	17	18	21
1	107	19	23	22	20	21	22	22
2	256	21	21	18	20	21	20	21
3	398	22	19	23	17	15	12	9
4	515	7	3	1	0	0	0	0
Size	526	79	78	78	72	74	72	73

Table 2. Index table, where the last row holds accumulated values (the size) for each group in number of address units.

do so by joining neighboring groups into supergroups. This is easy as long as all groups are written contiguously onto the same disk.

Because of the variable block size, we need an index to efficiently retrieve blocks when we read back one or more groups. The index is a table with one column for each group (subfile), and one row for each round in the Round Robin output process. The corresponding table element would then contain the size of each block. The size of the index table depends on the number of groups and the number of rounds. To speed up address calculations, it would be useful to add one column to the index table, holding the address of the first block in each round. It is shown in Table 4.3 how an index table might look like.

The size of each block could be in bytes, sectors or some other unit. The total size of each subfile after partitioning is found by adding the numbers in each column. This information will tell how many subfiles can be read together, or how much space is needed to hold the largest subfile during the final algebra operation.

Even for large data volumes, the index table should be stored in memory without eating up to much space: Suppose that for every round, approximately $2M$ data is written to disk. The size of the index table, in number of variable sized blocks is $s = \frac{Vp}{2M}$. If $V = 1$ GB, $p = 100$ and $M = 50$ MB, the size s is 1000 integers.

4.4 Estimation of the Free Space Limit

The free space limit should be as small as possible, however there are some constraints. If we have only one SCSI bus, we have no real I/O overlapping. An initiated I/O operation must be completed until another operation could be started. If we started a read operation and could not complete it because of lack of input buffer space, we would have a deadlock. Then we would not be able to free space by writing a group to disk. Before we start a read, we should make sure that we have enough free space to accommodate all data read.

4.5 The Minimum Block Size

Writing a group should be skipped if it is below a minimum size. This will happen only when the hashing formula works poorly, producing uneven sized groups, or

we try to stress the system, having uncoordinated parameters. The minimum block size should be in the vicinity of 4 KB, and this should be far below the normal group size when M has some reasonable value (above 100 KB, at least). If we use the LGF strategy there will be no minimum block size problem.

4.6 Multiuser Environments

It is also worth noting that the size of M can change dynamically, this only affects the size of the groups. Partitioning strategy, however, is done at operation startup time. Thus, if available memory decreases, an extra pass might be needed.

5 Cost Functions

Our cost model is based on I/O transfer only. This is the most significant cost factor, and in reasonable implementations, the CPU processing should go in parallel with I/O transfer making the CPU cost "invisible". Our disk model is traditional. Disk transfers are done blockwise. One block is the amount of data transferred in one I/O command. The main cost comes from two contributors: *the start up cost* and *transfer cost*. Start up cost comes from command software and hardware overhead, disk positioning time, and disk rotational delay.

In our model, the average start up cost is fixed, and is set equivalent to t_r, the time it takes to do one disk revolution. The transfer cost is directly proportional to the block size, and is equivalent to reading disk tracks contiguously, e.g., transfer cost is equal to $\frac{b}{V_s}t_r$, where V_s is the amount of data on one track and b is the block size to be transferred. For very large blocks, it is likely that several tracks and also tracks in different cylinders are read contiguously. This implies positioning, but we assume that the times used for this is insignificant compared to transfer time. The time it takes to transfer one block is $t_b = t_r(1 + \frac{b}{V_s})$. The time to transfer a data volume V using block size b is:

$$T_T = \frac{Vt_r}{b} + \frac{Vt_r}{V_s} = Vt_r\left(\frac{1}{b} + \frac{1}{V_s}\right) \tag{1}$$

Our emphasis is to find optimal parameters and the best working range for each method. To be able to handle the different equations mathematically we regard them as continuous functions rather than discontinuous functions resulting from applying ceiling and floor functions. This approximation gives a better overall view, but could cause small errors compared to the exact mathematical description.

To get smooth input and output streams, double buffering can be employed. However, to simplify the computations, the space for input buffers and extra output buffers is not counted within the memory M in our cost functions. Thus, the memory M, can be thought of as the available memory *after reservation of memory for the extra input and output buffers.*

The splitting is based on a hash formula applied to the operation key. The subtables will vary in size due to statistical variations and the "goodness" of the

selected hash formula, in a real system it would be beneficial to let the average subtable be slightly smaller than the workspace. With good approximation we can ignore this and we are also ignoring space needed for structural data.

5.1 One Pass Partitioning

We will start with a simple partitioning strategy. If the input table is larger than available work space, we will split the table in subtables, such that each subtable fits into work space M. The subtables are are written to temporary files. The necessary split factor is: $p = \lceil \frac{V}{M} \rceil$, and the time used is:

$$T_{1p} = 2T_T = 2Vt_r\left(\frac{1}{b} + \frac{1}{V_s}\right) = 2Vt_r\left(\frac{V}{M^2} + \frac{1}{V_s}\right)$$

The actual block size with fixed block size, assuming we are free to select the block size, is $b = \lfloor \frac{M}{p} \rfloor \approx \frac{M^2}{V}$, which gives:

$$T_{1p}^F = 2Vt_r\left(\frac{V}{M^2} + \frac{1}{V_s}\right)$$

and with variable block size, we get on the average $b = \frac{2M}{p} \approx \frac{2M^2}{V}$, which gives:

$$T_{1p}^V = 2Vt_r\left(\frac{V}{2M^2} + \frac{1}{V_s}\right)$$

For operand volumes slightly larger than M, we see that it would probably be better to let some of the work space be used for holding records participating in the final algebra stage, and use only a portion of the work space for split buffer. This leads us to the partial partitioning or hybrid hash algorithm. At the other end, we see that when operand volumes are very large, the split factor increases and the block size goes down. Small I/O blocks are severely slowing down the I/O process. It may be better to split the data repeatedly, using larger blocks, rather than using smaller blocks and reaching the correct subfile size in one pass. This leads us to the multipass splitting method.

5.2 Partial Partition

Partial partitioning is especially advantageous for operand volumes larger than M, but so small that we do not need all available memory for efficient splitting. A part of the memory is used for holding a number of complete groups, to avoid the unnecessary I/O caused by the splitting. If the operand volume V is known in advance, we can compute the optimum memory size x, not used for splitting. x is thus also the amount of data which is not split. We ignore floor and ceiling functions to keep the equations mathematically tractable.

Fixed Block Size. The number of subfiles is $N = (V - x)/M$. The complete split is done in one pass, hence $N = p$, the split factor. The average block size using fixed block size splitting is:

$$b = \frac{(M - x)}{p} = \frac{M(M - x)}{V - x}$$

which gives the split time by substitution of b into Eq. 1 is:

$$T_{pp}^{F} = 2(V - x)t_r\left(\frac{1}{b} + \frac{1}{V_s}\right) = 2(V - x)t_r\left(\frac{V - x}{M(M - x)} + \frac{1}{V_s}\right)$$

The time will vary with x, a large x reduces transferred volume, however, the average block size is decreased, and the time spent may increase. To find a minimal value for T_{pp}, we have to find the value of x that will give the minimum value of T_{pp}. This can be done by solving:

$$\frac{dT_{pp}^{F}(x)}{dx} = 2t_r\left(\frac{-2(V - x)M(M - x) + M(V - x)^2}{M^2(M - x)^2} - \frac{1}{V_s}\right) = 0$$

To solve this equation we substitute $\lambda = M/V_s$ and

$$z = \frac{V - x}{M - x} \tag{2}$$

Then we obtain the following equation in z giving:

$$z^2 - 2z - \lambda = 0 \Rightarrow z = 1 \pm \sqrt{1 + \lambda}$$

Back substitution of z into Eq. 2 gives:

$$x = \frac{zM - V}{z - 1}$$

x should never be negative, which implies: $z > 0$ and $zM - V \geq 0$. This sets a restriction on V, which must be less than zM. It does not make sense to use partial partitioning when operand volumes are greater than zM. When $V \leq M$, splitting is not at all employed. This limits the working range of partial partitioning to: $M < V \leq zM$.

Variable Block Size. Derivation in the case of variable block size is done the same way as for fixed block size. The only difference is that we are doubling the effective block size, which gives:

$$T_{pp}^{V} = 2(V - x)t_r\left(\frac{1}{b} + \frac{1}{V_s}\right) = 2(V - x)t_r\left(\frac{V - x}{2M(M - x)} + \frac{1}{V_s}\right)$$

5.3 Multipass Partitioning

Data are read from an input file and hashed into a number of subfiles. The actual block size with fixed size blocks is: $b = \lfloor \frac{M}{p} \rfloor \approx \frac{M}{p}$. The necessary number of subfiles at the end is $N = \lceil \frac{V}{M} \rceil \approx \frac{V}{M}$. Depending on the operand volume, it may be beneficial to partition the operand in several passes. This is equivalent to the merging used in sort-merge processing, and the approximate number of operand passes is $w = \log_p N$. When we substitute for N, we get $w = \log_p \frac{V}{M}$. The total amount of data read and written to disk to complete a partitioning and the final relational algebra step is $V_T = 2Vw$, which gives a total partitioning time of:

$$T_{mp} = V_T T_T = 2Vt_r \log_p \frac{V}{M}\left(\frac{1}{b} + \frac{1}{V_s}\right)$$

Substituting for $\log_p \frac{V}{M} = \frac{\ln V/M}{\ln p}$:

$$T_{mp} = 2V\frac{\ln V/M}{\ln p}t_r\left(\frac{p}{M} + \frac{1}{V_s}\right) \tag{3}$$

An optimum block size exist because when we split into many subfiles in one pass, each subfile need an output buffer, hence they get smaller. We can minimize Eq. 3 by noting that the variable part of this expression is:

$$f(p) = \frac{1}{\ln p}\left(\frac{1}{b} + \frac{1}{V_s}\right)$$

To find a minimum value for $f(p)$, we derive $f(p)$ with respect to p, and the optimum value for p is when $f'(p) = 0$:

$$p(\ln p - 1) - \frac{M}{V_s} = 0$$

It is easily seen that p is a function of the quotient $\lambda = \frac{M}{V_s}$:

$$p(lnp - 1) - \lambda = 0$$

This equation can be solved numerically, to find the value for p. Thus, the function for multipass partitioning with fixed block size is given by:

$$T^F_{mp} = 2Vt_r \log_p \frac{V}{M}\left(\frac{p}{M} + \frac{1}{V_s}\right)$$

With the same method, we can find an optimum value for p in the case of variable block size (where $b = \frac{2M}{p}$):

$$T^V_{mp} = 2Vt_r \log_p \frac{V}{M}\left(\frac{p}{2M} + \frac{1}{V_s}\right)$$

As an example, Table 5.3 shows optimal split factors, block size, and number of passes for different sizes of memory, with operand volume held constant on $V = 1000$ MB and $V_s = 50$ KB.

$\lambda = M/V_s$: (M in MB:)	1 (0.05)	2 (0.1)	5 (0.25)	10 (0.5)	20 (1.0)	50 (2.5)	100 (5)	200 (10)	500 (25)	1000 (50)
Fixed block size:										
Optimal p	3	4	6	8	12	23	37	63	129	226
b	17	25	42	62	83	109	135	159	194	226
w	9.01	6.64	4.63	3.66	2.78	1.91	1.47	1.11	1.00	1.00
Var. block size:										
Optimal p	4	5	8	12	20	37	63	108	226	400
b	25	40	62	83	100	135	159	185	221	250
w	7.14	5.72	3.99	2.31	1.66	1.28	1.00	1.00	1.00	1.00

Table 3. Optimal split factors, block size and number of passes for different sizes of memory.

Strategy:	Partial	One Pass	Multipass
Range:	$M < V \leq zM$ (medium operands)	$zM < V \leq p_{mp}M$ (large operands)	$V > p_{mp}M$ (huge operands)
Fixed Size, T^F: $z = 1 + \sqrt{1+\lambda}$ $p(\ln p - 1) - \lambda = 0$	$2(V-x)t_r\left(\frac{V-x}{M(M-x)} + \frac{1}{V_s}\right)$	$2Vt_r\left(\frac{V}{M^2} + \frac{1}{V_s}\right)$	$2Vt_r \log_p \frac{V}{M}\left(\frac{p}{M} + \frac{1}{V_s}\right)$
Variable size T^V: $z = 1 + \sqrt{1+2\lambda}$ $p(\ln p - 1) - 2\lambda = 0$	$2(V-x)t_r\left(\frac{V-x}{2M(M-x)} + \frac{1}{V_s}\right)$	$2Vt_r\left(\frac{V}{2M^2} + \frac{1}{V_s}\right)$	$2Vt_r \log_p \frac{V}{M}\left(\frac{p}{2M} + \frac{1}{V_s}\right)$
Supporting values: $\lambda = M/V_s$	$x = \frac{zM-V}{z-1}$		$w = max\left(1.0, \frac{\ln V/M}{\ln p}\right)$

Table 4. Cost functions for splitting.

5.4 Summary and Application of the Cost Functions

The cost functions and their working areas are summarized in Table 5.4. To demonstrate the relationship between the three variants, we have computed the partitioning times for different operand volumes. By using $V_s = 50$ KB, $t_r = 11$ ms, and $M = 1$ MB, we get the partitioning times as shown in Table 5.4.

From the table, we can see that for fixed size blocks, partial partitioning is best for volumes from 2 to 5 MB. Multipass is better when operands are larger than 12 MB. For variable blocks, partial partitioning is best for the volumes from 2 to 7 MB. Then one pass takes over, multipass partitioning is better when operands are larger than 20 MB. The different working areas for the variants of partitioning is illustrated in Fig. 3.

	Fixed sized blocks			Variable sized blocks		
	$\lambda = 20, z = 5.6, p_{mp} = 12$			$\lambda = 20, z = 7.4, p_{mp} = 20$		
V (in MB)	T_{pp}	T_{1p}	T_{mp}	T_{pp}	T_{1p}	T_{mp}
2	0.69	0.98	1.39	0.61	0.93	1.33
3	1.39	1.53	2.19	1.22	1.43	2.00
4	2.08	2.13	2.92	1.83	1.96	2.37
5	2.77	2.78	3.65	2.44	2.50	3.33
6	-	3.47	4.38	3.04	3.07	4.00
7	-	4.20	5.11	3.65	3.66	3.67
8	-	4.98	5.84	-	4.27	5.34
16	-	12.8	12.7	-	9.96	10.7
32	-	37.0	31.7	-	25.6	24.7
64	-	119.5	76.1	-	74.0	59.2
128	-	421.0	177.6	-	238.9	138.2
256	-	1570.0	405.9	-	842.0	315.9

Table 5. Partitioning times for different operand volumes and different basic partitioning methods.

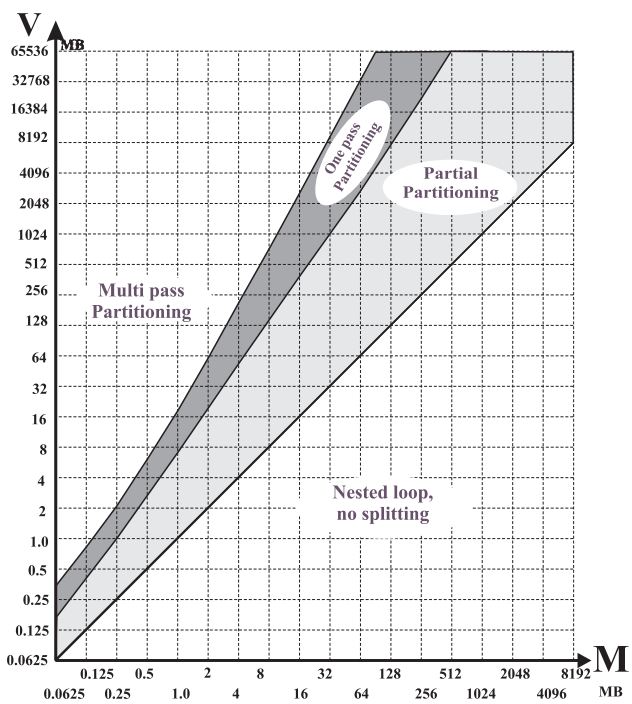


Fig. 3. Working areas for different algebra methods, with numbers from variable block size.

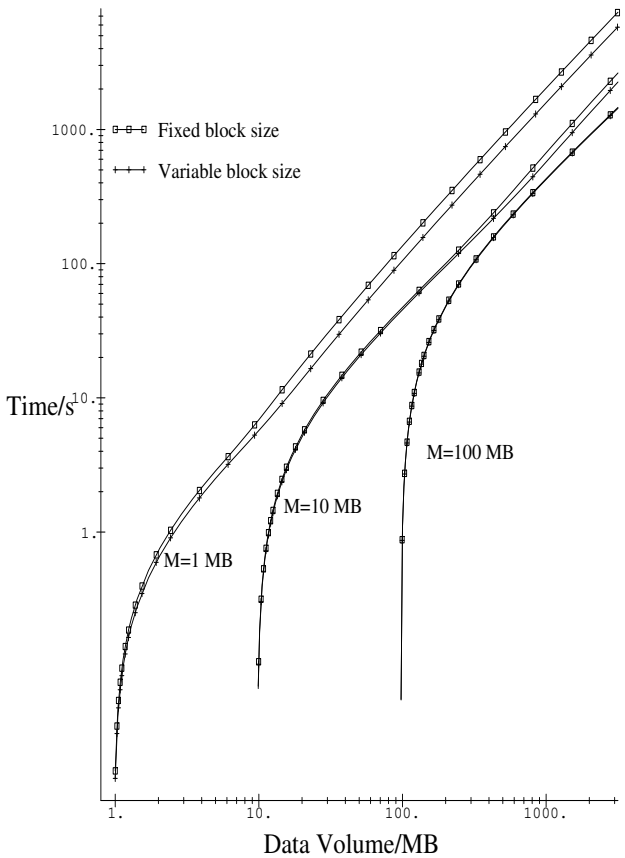


Fig. 4. Execution time with snow plow splitting.

6 Comparison

To compare the effect of increased average block size, we compare the traditional and new methods for three representative cases: $M = 1$ MB, $M = 10$ MB, and $M = 100$ MB. In Fig. 4 we have plotted the functions for variable and fixed block size. We have for each value of V used the partitioning strategy (no/partial/one pass/multipass) that gives the best time. It is important to keep in mind that this is a log-log-plot, and therefore the difference in execution time between fixed and variable block size is not large in the figure. The improvement can be better illustrated by looking at the performance gain, which is computed in Table 6 and illustrated in Fig. 5. In the table, blank fields denotes no splitting.

As expected, the improvements are most noticeable for smaller work space areas. For larger work spaces, the number of blocks is relatively small, and the block access time is negligible compared to the transfer time.

	$M = 1$ MB, $\lambda = 20$			$M = 10$ MB, $\lambda = 200$			$M = 100$ MB, $\lambda = 2000$		
V (in MB)	fixed	var	imp %	fixed	var	imp %	fixed	var	imp %
1									
2	0.7	0.6	14						
4	2.1	1.8	14						
8	5.0	4.3	17						
16	12.7	10.0	27	3.1	2.9	4.2			
32	31.7	24.7	29	11.3	10.8	4.2			
64	76.1	59.2	29	27.6	26.5	4.2			
128	177.6	138.2	29	60.4	58.0	4.2	13.0	12.8	1.3
256	405.9	315.9	29	128.3	121.1	6.0	72.5	71.6	1.3
512	913.2	710.8	29	285.8	256.7	11.3	191.5	189.2	1.3
1024	2029	1580	29	668.6	571.6	17.0	429.4	423.6	1.3
2048	4464	3475	29	1538	1314	17.0	905.4	893.2	1.3

Table 6. Performance improvement as a function of operand volume and workspace size.

7 Conclusions and Future Work

We have described a novel method for partitioning, the circular track snow plow buffer management strategy. This strategy makes more efficient use of memory compared to traditional methods, and effectively doubles the average block size. Doubling the effective block size can give substantial reductions in I/O transfer time, and by the use of cost models we have shown that performance gains of 10-20% can be expected for reasonable resource configurations.

In this paper we have made the assumption that the operand volume is known. This is not always true, and we are now investigating a dynamic or adaptive strategy, based on the snow plow principle, which is applicable when the operand volume is unknown. We are also working with the new buffer management strategy employed in other areas as file loading and transaction log.

In the development og this methods, we have also discovered how a more efficient set of routines for communication with the SCSI buss system could be used to avoid the unnecessary transfer of data to an internal block buffer. Direct transfer to the user program area can save the internal bus from considerable traffic. This is clearly interesting, and should be studied further.

References

1. K. Bratbergsengen. Hashing Methods and Relational Algebra Operations. In *Proceedings of the 10th International Conference on VLDB*, 1984.
2. K. Bratbergsengen, R. Larsen, O. Risnes, and T. Aandalen. A Neighbor Connected Processor Network for Performing Relational Algebra Operations. In *Fifth Workshop on Computer Architecture for Non-Numeric Processing, March 11-14, 1980 (SIGIR Vol. XV No. 2, SIGMOD Vol.X No. 4)*, 1980.
3. D. L. Davidson and G. Graefe. Memory-Contention Responsive Hash Joins. In *Proceedings of the 20th International Conference on VLDB*, 1994.

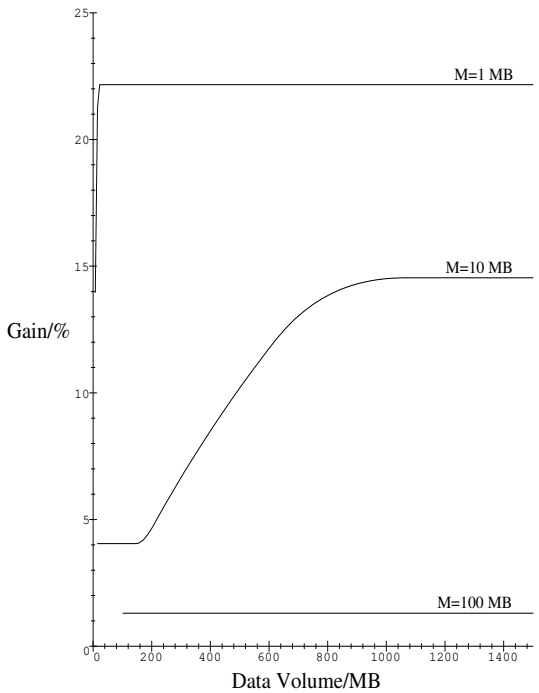


Fig. 5. Performance gain by using the snow plow strategy.

4. D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. ACM SIGMOD Conf.*, 1984.
5. G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), 1993.
6. G. Graefe. Volcano — An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1), 1994.
7. M. Kitsuregawa, H. Tanaka, and T. Motooka. Application of Hash to Data Base Machine and its Architecture. *New Generation Computing*, 1(1), 1983.
8. D. Knuth. *The Art of Computer Programming. Sorting and Searching.* Addison-Wesley Publishing Company Inc., 1973.
9. M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-Partitioned Join Method Using Dynamic Destaging Strategy. In *Proceedings of the 14th International Conference on VLDB*, 1988.
10. L. D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3), 1986.
11. H. Zeller and J. Gray. An Adaptive Hash Join Algorithm for Multiuser Environments. In *Proceedings of the 16th International Conference on VLDB*, 1990.

This article was processed using the LaTeX macro package with LLNCS style

Appendix C

An Analytical Study of Object Identifier Indexing

This appendix contains the paper presented at the 9th International Conference on Database and Expert Systems Applications (DEXA'98), held in Vienna, Austria, August 1998.

An Analytical Study of Object Identifier Indexing

Kjetil Nørvåg and Kjell Bratbergsengen

Department of Computer and Information Science
Norwegian University of Science and Technology
7034 Trondheim, Norway
{noervaag,kjellb}@idi.ntnu.no

Abstract. To avoid OID index retrieval becoming a bottleneck, efficient buffering strategies are needed to minimize the number of disk accesses. In this paper, we develop analytical cost models which we use to find optimal sizes of the index page buffer and the index entry cache, for different memory sizes, index sizes, and access patterns. Because existing buffer hit estimation models are not applicable for index page buffering in the case of tree based indexes, we have also developed an analytical model for index page buffer performance. The cost gain from using the results in this paper is typically in the order of 200-300%. Thus, the results should be of valuable use in optimizers and tools for configuration and tuning of object-oriented database systems.

1 Introduction

In a large OODB with logical object identifiers (OIDs), the OID index (OIDX) can be quite large, typical in the order of 20% of the size of the database itself [7]. This means that in general, only a small part of the OIDX fits in main memory, and that OIDX retrieval can become a bottleneck if efficient access and buffering strategies are not applied. As the amount of memory increases, most of the frequently accessed part of the index and database will fit in main memory, and in this case, it is the cost of the *infrequently accessed* data and index structures that will represent the disk bottleneck. In this paper, we will study OIDX retrieval analytically, and use the results to reduce the average number of disk accesses needed to retrieve an index entry, in many cases to only a fraction of the original cost.

Traditionally, the most recently used *index pages* have been kept in the buffer pool to make OID mapping efficient, usually managed as an LRU chain. However, if the index entries have low locality, which is often the case for OID indexes, only a small part of the information in the pages in the buffer is really being used. To better utilize the memory, it is possible to keep the most recently used *index entries* in an OID entry cache (OE cache), as is done in the Shore OODB [11]. This can improve performance considerably, but it is important to know how much of the memory should be used for buffering index pages,

and how much should be used for caching[1] index entries. This issue has not previously been studied in detail. The OIDX access cost is also an issue that has been underestimated in research literature, the OID mapping cost have not been included into the cost models. As is evident from the results presented later in this paper, this cost can not be ignored, and we will later show that there is much to be gained by optimization.

An OIDX is usually realized as a hash file or a B-tree. Based on simulation results from a comparison study by Eickler et al. [7], and our own study of OID indexing, we believe B-trees or ISAM variants to be the best suited for OID indexing, and therefore we limit this discussion to tree based indexing. It should be noted, however, that much of this paper is still relevant to hash file indexing, the only difference is the index access equations, which are simpler for hash files.

The organization of the rest of the paper is as follows. Sect. 2 gives an overview of related work. In Sect. 3 we describe our OODB buffer model. In Sect. 4 and Sect. 5 we describe our assumptions regarding access pattern and our page access model. In Sect. 6 we present the Bhide, Dan and Dias LRU buffer model (BDD model) [2], which our index buffer model in Sect. 7 is based on. In Sect. 8 we develop an OID retrieval cost model, and in Sect. 9 we study how different memory and index sizes affects the performance. Finally, in Sect. 10, we conclude the paper.

2 Related Work

There have been several approaches to estimate the number of page accesses in the case of non-hierarchical files [3, 13, 14], and studies of the characteristics and validity of these estimates [6, 9]. However, they have not taken access pattern and buffering into account. Accesses have been assumed to be uniform, and an infinite buffer has been assumed, where only the first access to a page is counted.

Recently, other approximations have been developed, the most interesting is the work done by Bhide, Dan and Dias [2], which is an improvement of a model presented by Dan and Towsley [5]. They model requests through an LRU buffer, and the model is valid under the assumption that each request is independent of all previous requests. We will describe this model in more detail in Sect. 6.

Modeling buffer in the case of hierarchical files, where traversing the tree makes the request independency assumption invalid, complicates the situation. One modeling approach is presented in [10], where an I/O model for index scans with LRU buffer is presented. However, this model assumes a complete scan of the index, and is thus a model for uniform access to the index, all entries have the same probability of access (essentially one access during one scan).

Several cost models for OODBs have been developed, for example [1, 8], but they do not include buffer and OID mapping cost.

[1] In this paper, we use buffer and buffering when we talk about index *pages* in memory, and cache and caching when we talk about index *entries* in memory. This is established terminology, but apart from the naming, there is no real difference between buffering and caching techniques in this case.

3 Buffer

The buffer in a page server OODB is used for buffering object pages, index pages, and optionally, index entries, as illustrated on Fig. 1. In the rest of this paper, we will denote the index page buffer size as M_{ipages}, the OE cache size as M_{ocache}, and the sum of these as M_i, as illustrated.

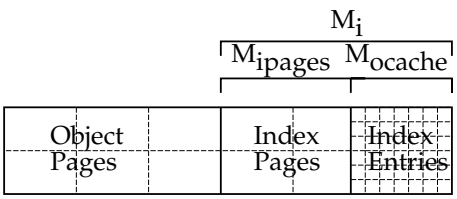


Fig. 1. Buffer in a page server OODB.

In a real implementation, there will usually not be separate buffer areas for object and index pages. All index and data pages resides in a common buffer, managed by some buffer management policy, for example DBMIN [4]. With this buffer management policy, it is possible to put restrictions on how much of the buffer a certain task/transaction can allocate. This is necessary to avoid situations where some operation do something that replaces all the pages in the buffer, e.g. a scan operation. With such a management strategy, it is also possible to control the amount of memory used for indexing, and even without such policies, it is still easy to know the amount of buffer occupied by index pages by simple bookkeeping.

The size of the OE cache can be fixed (set at system startup time), or adaptive. We will later in this paper study how an adaptive OE cache can improve performance over a fixed size OE cache.

4 Index Entry Access Model

We assume accesses to objects in the database system to be random, but skewed (some objects are more often accessed than others). We assume it is possible to (logically) partition the range of OIDs into partitions, where each partition has a certain size and access probability. This is illustrated on the bottom of Fig. 2. We consider a database in a stable condition, with N_{obj} objects (and hence, N_{obj} index entries).

In many analysis and simulations, the 80/20 model is applied, where 80% of the accesses go to 20% of the database. While this is satisfactory for analysis of some problems, it has a major shortcoming when used to estimate the number of distinct objects to be accessed. When applied, it gives a much higher number of distinct accessed objects than in a real system. The reason is that for most applications, inside the hot spot area (20% in this case), there is an even hotter and smaller area, with a much higher access probability. This has to be reflected in the model.

In the simulations and analysis described in this paper, we have employed two different partition sets, each with three partitions (note that this is accesses to OIDs, and not to pages in the index). In both partitioning sets, the 20% hot spot area from the 80/20 model is partitioned. In the first partitioning set, we

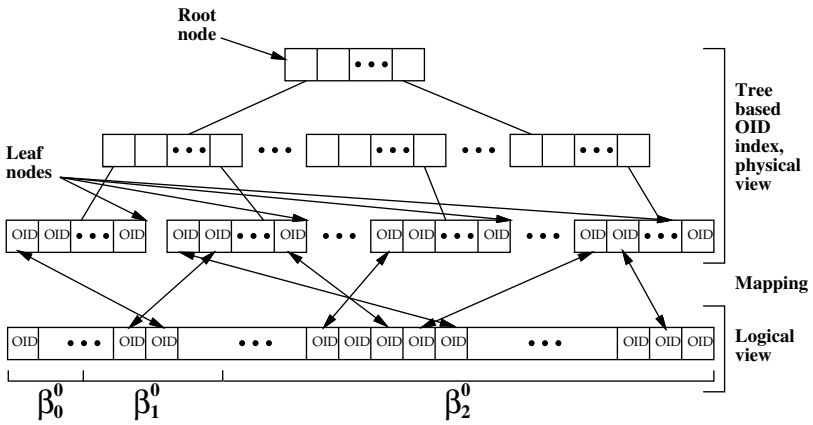


Fig. 2. OID index. The lower part shows the index from a logical view, the upper part is as an index tree, which is how it is realized physically. We have indicated with arrays how the entries are distributed over the leaf nodes.

Partition Size, Set 1	Partition Size, Set 2	Partition Access Probability
$\beta_0^0 = 0.01$	$\beta_0^0 = 0.001$	$\alpha_0^0 = 0.64$
$\beta_1^0 = 0.19$	$\beta_1^0 = 0.199$	$\alpha_1^0 = 0.16$
$\beta_2^0 = 0.80$	$\beta_2^0 = 0.80$	$\alpha_2^0 = 0.20$

Table 1. Partition sizes and partition access probabilities for two partitioning sets used in the study.

have a 1% hot spot area, in the other, a 0.1% hot spot area. The partition sizes and partition access probabilities are summarized in Table 1. β_i^0 denotes the size of partition i, as a fraction of the total database size, and α_i^0 denotes the fraction of accesses done to partition i.

5 Index Page Access Model

As noted, we can assume low locality in index pages. Because of the way OIDs are generated, entries from a certain partition are not clustered in the index. This is illustrated in Fig. 2, where a leaf node containing index entries contains unrelated entries from different partitions. This means that the access pattern for the leaf nodes is different from the access pattern to the database from a logical view.

We use the initial index entry partitioning (the index entry access pattern) as basis for deriving the index page partitioning (the index page access pattern). With a totally unclustered index, the index entries from the different partitions are distributed over the leaf nodes with binomial distribution. To simplify this analysis, we make some approximations and assumptions: The second most hot area has a number of entries sufficiently larger than the number of leaf nodes,

i.e., $\beta_1^0 > \frac{1}{F}$. This is a reasonable assumption, with a page size of 8 KB, this gives $\beta_1^0 > 0.002$. We assume that the accesses to entries not belonging to the hottest hot spot can be modeled as random and uniform, and it is the hottest hot spot area alone that decides the index page access pattern. Further, we approximate the binomial distribution by assuming the index entries are distributed over the leaf nodes with uniform distribution. Simulation results show that the error is acceptable.

We consider two cases: 1) the number of hot spot entries is smaller than the number of leaf nodes, $\beta_0^0 N_{obj} < N_{\text{tree}}^0$, and 2) the number of hot spot entries is larger than the number of leaf nodes, $\beta_0^0 N_{obj} > N_{\text{tree}}^0$.

Case 1: The number of hot spot entries is smaller than the number of leaf nodes. When the number of objects in the hot spot area is less than the number of leaf nodes, pages belong to one of two partitions: Those that have an hot spot entry, and those which have not:

$$\begin{array}{ll} \beta_{L0} = \beta_0^0 N_{obj}/N_{\text{tree}}^0 & \alpha_{L0} = \alpha_0^0 + \beta_{L0} \sum_{i=1}^{p} \alpha_i^0 \\ \beta_{L1} = 1 - \beta_{L0} & \alpha_{L1} = 1 - \alpha_{L0} \end{array}$$

Case 2: The number of hot spot entries is larger than the number of leaf nodes. In the case where there is one or more hot entry in each page, we will with a uniform distribution over the pages have some of the pages with one hot index entry more than the others. Especially in the case where there are few hot index entries in each page, it is important to capture this fact in the model. We now have two partitions, one with the pages containing that extra index entry, and the other with those pages that do not:

$$\begin{array}{ll} \beta_{L0} = (\beta_0^0 N_{obj} - \lfloor \beta_0^0 N_{obj}/N_{\text{tree}}^0 \rfloor N_{\text{tree}}^0)/N_{\text{tree}}^0 & \alpha_{L0} = \beta_{L0} N_{\text{tree}}^0 \frac{\lceil \beta_0^0 N_{obj}/N_{\text{tree}}^0 \rceil}{\beta_0^0 N_{obj}} \\ \beta_{L1} = 1 - \beta_{L0} & \alpha_{L1} = 1 - \alpha_{L0} \end{array}$$

With an increasing number of hot spot index entries in each page, the access pattern will get more and more uniform.

6 The BDD LRU Buffer Model

In our analysis, we need to estimate the buffer hit probability in an LRU managed buffer. We do this with the BDD LRU buffer model [2]. We will only briefly explain the model in this section, the derivation and details behind the equations can be found in [2]. A database in the BDD model has of size N data granules (pages or objects), partitioned into p partitions. Each partition contains β_i of the data granules, and α_i of the accesses are done to each partition. The distributions *within* each of the partitions is assumed to be uniform. All accesses are assumed to be independent.

After n accesses to the database, the number of distinct data granules (pages, objects, or index entries) from partition i that have been accessed is:

$$B_i(n) = \beta_i N(1 - (1 - \frac{1}{\beta_i N})^{\alpha_i n})$$

When the number of accesses n is such that the number of distinct data granules accessed is less than the buffer size B, $\sum_{i=1}^{p} B_i(n) \leq B$, the buffer hit probability for partition i is:

$$P_i(n) = 1 - (1 - \frac{1}{\beta_i N})^{\alpha_i n}$$

and the overall buffer hit probability is:

$$P(n) = \sum_{i=1}^{p} \alpha_i P_i(n)$$

The steady state average buffer hit probability can be approximated to the buffer hit ratio when the buffer becomes full, i.e., n is chosen as the largest n that satisfies $\sum_{i=1}^{p} B_i(n) \leq B$, where B in this case is the number of data granules that fits in the buffer:

$$P_{\text{buf}}(B, N) = P(n) \tag{1}$$

7 General Index Buffer Model

In the previous section, we presented the BDD LRU buffer model for independent, non-hierarchical, access. Modeling buffer for hierarchical access is more complicated. Even though searches to the leaf page can be considered to be random and independent, nodes accessed during traversal of the tree are *not* independent. We will in this section present a general index buffer model, where the granularity for access is a page.

In the generic tree used in this model, the size of one index page is S_P, and the size of one index entry S_{ie}. This give a fanout $F = \lfloor S_P/S_{\text{ie}} \rfloor$ and with a database consisting of N_{obj} objects to be indexed, the number of leaf nodes is $N^0_{\text{tree}} \approx \frac{N_{\text{obj}}}{\lfloor S_P/S_{\text{ie}} \rfloor}$.

Our approach to approximate the buffer hit probability, is based on the obvious observation that *each level* in the tree is accessed with the *same probability* (assuming traversal from root to leaf on every search). Thus, with a tree where the index nodes has a fanout F, the number of levels in the tree is $H = 1 + \lceil \log_F N^0_{\text{tree}} \rceil$. We initially treat each level in the tree as one partition, thus, initially we have H partitions. Each of these partitions is of size N^i_{tree}, where N^i_{tree} is the number of index pages on level i in the tree. The access probability is $\frac{1}{H}$ for each partition.

To account for hot spots, we further divide the leaf page partition into p' partitions, each with a fraction of β_{Li} of the leaf nodes, and access probability α_{Li} relative to the other leaf page partitions. Thus, in a "global" view, each of

these partitions have size $\beta_{Li}N^0_{\text{tree}}$ and access probability $\frac{\alpha_{Li}}{H}$. In total, we have $p = p' + (H-1)$ partitions. The hot spots at the leaf page level make access to nodes on upper levels non-uniform, but as long as the fanout is sufficiently large, and the hot spot areas are not too narrow, we can treat accesses to nodes on upper levels as uniformly distributed within each level. With the modifications described, a tree of height H, and p' leaf page partitions, the equations for α_i and β_i (access probability and fractional size of each partition) becomes:

$$\alpha_i = \begin{cases} \frac{\alpha_{Li}}{H} & \text{if } i < p' \\ \frac{1}{H} & \text{if } p' \leq i < p \end{cases} \qquad \beta_i = \begin{cases} \frac{\beta_{Li}N^0_{\text{tree}}}{N_{\text{tree}}} & \text{if } i < p' \\ \frac{N^i_{\text{tree}}}{N_{\text{tree}}} & \text{if } p' \leq i < p \end{cases}$$

We denote the number of leaf nodes as N^0_{tree}. The total number of nodes in the tree, N_{tree}, can then be calculated as:

$$N_{\text{tree}} = \sum_{i=0}^{H-1} N^i_{\text{tree}} \quad \text{where} \quad N^i_{tree} = \left\lceil \frac{N^{i-1}_{\text{tree}}}{F} \right\rceil$$

We denote the overall buffer probability, by using the BDD buffer hit probability equation (Equation (1)) with α_i and β_i as defined above as $P_{\text{buf_ipage}}(B, N_{\text{tree}})$ where $B = \lfloor \frac{M_{\text{ipages}}}{S_P} \rfloor$ is the buffer size, and N_{tree} is the total number of index pages in the tree.

To validate the analytical model, we have compared it with simulation results. The simulations have been performed with different index sizes, buffer sizes, index page fanouts, and access patterns. Typical deviation between model and simulations is much smaller than 1%. More details about the validation results can be found in [12].

8 An OID Retrieval Cost Model

Our OID retrieval cost model is based on the number of disk page accesses, since this is the most significant cost factor and a common bottleneck. Disk I/O is only necessary if the requested information can not be found in memory. The model includes an index page buffer, and an OE cache. It is important to note that in the general case, with an ordinary data index, an OE cache can in some situations have a very costly side effect. An increase in the OE cache size reduces the number of complete pages in the buffer, something that can make lots of costly index page installation reads necessary when the entries in the index is to be updated. In an OIDX, this issue is not so important. Index entry modification does not happen often, usually only because of object migration or schema change. A tree based OIDX is essentially an append-only structure, the only potentially troublesome operation is deletion of objects, but index modifications because of this can be done in batch and/or as a background activity.

8.1 OID Entry Cache

A certain amount of memory, M_{ocache}, is reserved for the OE cache. If we assume the size of each entry is S_{ie}, and an overhead of S_{oh} bytes is needed for each entry, the number of entries that fits in the OE cache is approximately $N_{\text{ocache}} \approx \frac{M_{\text{ocache}}}{(S_{\text{ie}}+S_{\text{oh}})}$. Accesses to the OE cache can be assumed to follow the assumptions behind the BDD model, they are independent random requests, and by applying this model with object entries as data granules, we estimate that the probability of an OE cache hit is $P_{\text{ocache}} = P_{\text{buf}}(N_{\text{ocache}}, N_{\text{obj}})$, with p' partitions as defined in Sect. 7.

The results in the following analysis are highly dependent of the amount of overhead S_{oh} needed for each entry. Of course, with a minimal overhead, it would always be beneficial to use as much as possible of the total index memory as an OE cache. The most reasonable way to implement it, and at the same time keep the CPU cost low, is to use an hash table to provide fast access to the entries. In that case, approximately two pointers are needed for each entry on average.

8.2 Total OID Retrieval Cost

With a probability of P_{ocache}, the OID entry requested is already in the OE cache, but for $(1 - P_{\text{ocache}})$ of the requests, we have to access the index pages, and one or more disk accesses might be needed. The probability of a given index page being in memory is *on average* $P_{\text{buf_ipage}}$, with α_{Li} and β_{Li} as defined in Sect. 5. To traverse the H levels from the root to a leaf node, we need $(1 - P_{\text{buf_ipage}})H$ disk accesses. The average cost (number of disk accesses) of an index lookup is $C_{\text{indexlookup}} = (1 - P_{\text{ocache}})(1 - P_{\text{buf_ipage}})H$.

9 Optimizing OE Cache and Buffer Sizes

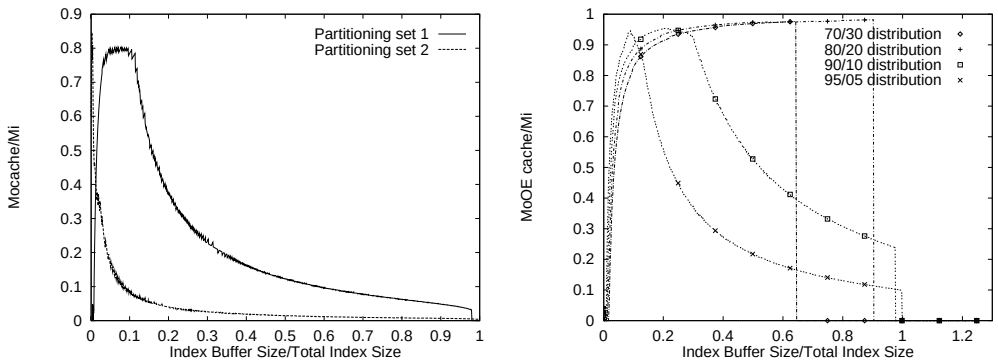
We have now derived the equations necessary to calculate the cost of an OIDX lookup. We will in this section study in detail under which conditions an OE cache is beneficial, and how much of the index memory should be reserved as an OE cache, to get optimal performance.

The optimization problem is, given a certain amount of index memory,[2] to find the combination of M_{ipages} and M_{ocache} that gives the lowest cost, given the invariant $M_i = M_{\text{ocache}} + M_{\text{ipages}}$. In the rest of this paper, we give OE cache size as a fraction of the total index memory size.

The study was done with databases of two different sizes: DB1 with 2 million objects, and DB2 with 20 million objects. *The optimal OE cache sizes were the same for both database sizes.* Unless otherwise noted, results and numbers in the next sections are from the study of DB1, with an index page size of 8 KB, and access pattern according to partitioning set 1 (Table 1).

[2] Note that memory needed for buffering of data pages is not included, this issue is orthogonal to the one studied here.

9.1 Optimal OE Cache Size Versus Index Size



(a) Three partitions. (b) Two partitions.

Fig. 3. Optimal OE cache size vs. amount of index memory available.

Figure 3a shows the optimal M_{ocache} for different memory sizes, where optimal M_{ocache} is the one that minimizes $C_{\text{indexlookup}}$. An interesting observation, is that the optimal M_{ocache} can be relatively large, especially for medium buffer sizes.

The OE cache is most useful for capturing the accesses to the hot and medium hot areas. This is evident from the figures. As the memory size increases beyond the size needed to capture these areas, the relative fraction of index memory useful for OE cache decreases.

When the buffer size is very small, and only a small part of the hot set fits in it, most entries in the OE cache will only be accessed once each time they are brought into the OE cache. The result is that reserving a large part of the little memory available for the OE cache can be a waste of space. In addition, to reduce retrieval cost, it will be beneficial to have as many as possible of the upper levels of the index tree in the index buffer. When the size of the index memory increases, we come to a point where most of the hot set fits in the OE cache, in addition to the most frequently accessed pages fits in the index page buffer. It is with these memory sizes (relative to the index size) we should have the largest OE cache sizes. If we increase memory even more, most of the hot set fits in the OE cache, and it is advantageous to use the extra memory for index pages. The relative size of the OE cache decreases (but the absolute size stays the same, or increases). As memory increase even more, we come to a point where the optimal OE cache size drops to zero. The reason for this, is that storing the index entries as separate entries instead of pages, incur a relatively high overhead, in the order of 50%. Stored as pages, a higher number of entries fits in the index memory,

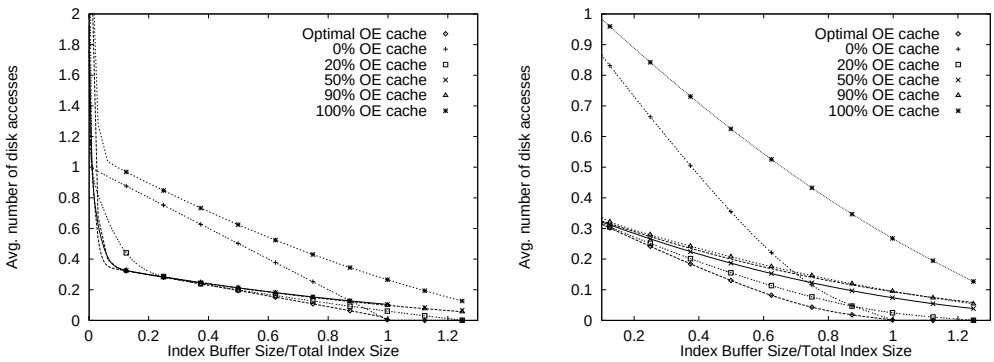


Fig. 4. Cost, partitioning set 1 to the left, partitioning set 2 to the right.

and when we come to a certain point, we get a higher buffer hit probability by storing the index entries as index pages instead of having them in the OE cache.

9.2 Mapping Cost with Different OE Cache Sizes

Figure 4 presents the average mapping costs for partitioning sets 1 and 2. It illustrates the benefits of using OE caching. It also emphasizes the importance of using the *optimal* OE cache size. It can be seen from the figure, that even though choosing a constant size of for example 50% is safe in general, if optimal performance is desired, the OE cache size should be adaptively tuned according to memory and index size. An important fact is that for most memory sizes, almost any OE cache size is better than using no OE cache at all (0% OE cache in the figures).

9.3 The Effect of Different Access Patterns

The optimal OE cache size can be highly dependent of the access pattern. Figure 3a illustrates this for the two partitioning sets. We see that as the most narrow hot spot area (0.01%) gets more narrow (0.001%), the area where a large OE cache is needed also gets smaller.

To be able to study the effect of different access patterns, we have done some studies with only two partitions, to see how the changing sizes and access probabilities affects the optimal cache size and OID retrieval cost. Figure 3b shows how the optimal OE cache size changes with different distributions. It is interesting to note that with a wide hot spot area, e.g. 70/30, it is beneficial to use almost all the available index memory for OE cache, up to a certain point. At this point, the optimal cache size drops to zero.

In the previous section, with three partitions, we had a much smaller hot spot area than in this case. The result was a drop in optimal OE cache size relative to total index memory when the OE cache was large enough to keep the hot spot, with a decline until the point where the buffer hit probability was higher with only index pages. Here we get the same result with 95/05 and 90/10

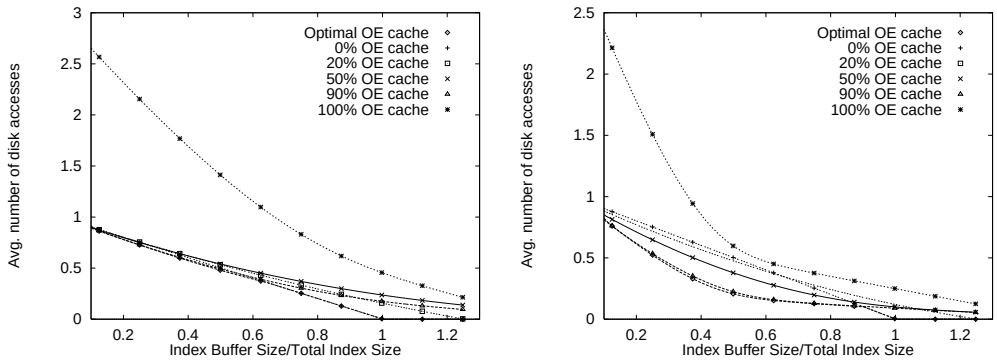


Fig. 5. Cost, 70/30 distribution of accesses to the left, 80/20 distribution of accesses to the right.

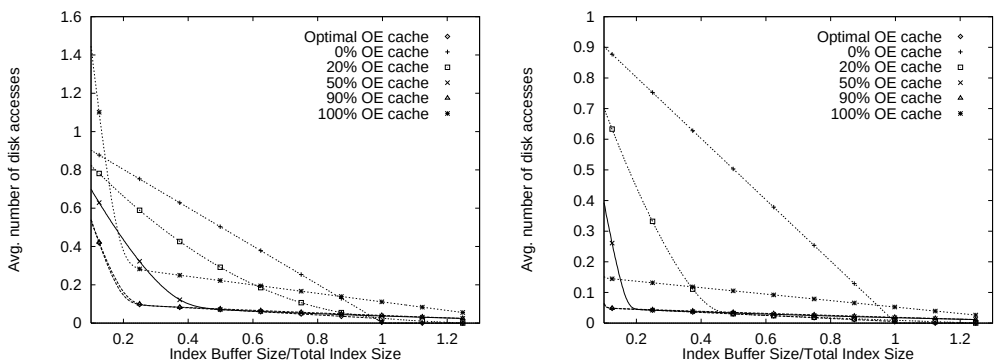


Fig. 6. Cost, 90/10 distribution of accesses to the left, 95/05 distribution of accesses to the right.

distributions, but not with the distributions with wider hot areas. The reason is that with such wide hot areas, the leaf page access distribution is almost uniform.

In Fig. 5 and 6 the mapping cost is shown for various OE cache sizes. They show how important it is to have an optimal OE cache size, and that a fixed size OE cache typically make the OID mapping process 200-300% more costly than necessary.

10 Conclusions

In this paper, we have developed an analytical model for OIDX retrieval cost which includes index buffer and OE caching. As a part of this work, we have also developed an analytical model for hierarchical index buffer performance, based on the LRU model of Bhide et al. [2]. We used the OIDX retrieval cost model to study how memory size, index size, access patterns and different OE cache sizes affects buffer hit performance. This showed that:

1. The relative amount of memory that should be reserved for OE caching is mainly dependent of the size of the index relative to the size of the index memory, not their absolute values.
2. The relative amount of memory that should be reserved for OE caching is highly dependent of the access pattern.
3. The gain from using the optimal size of the OE cache can be very high. This is very important as the size of the available memory increases. When most of the frequently used data and index structures fits in the buffer, it is the requests for the infrequently accessed items, that is not resident in memory when requested, that will use most of the disk bandwidth.

References

1. E. Bertino and P. Foscoli. On modeling cost functions for object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3), 1997.
2. A. K. Bhide, A. Dan, and D. M. Dias. A simple analysis of the LRU buffer policy and its relationship to buffer warm-up transient. In *Proceedings of the Ninth International Conference on Data Engineering*, 1993.
3. A. F. Cardenas. Analysis and performance of inverted data base structures. *Communications of the ACM*, 18(5), 1975.
4. H. T. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 14th VLDB Conference*, 1985.
5. A. Dan and D. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proceedings of ACM SIGMETRICS*, 1990.
6. G. Diehr and A. N. Saharia. Estimating block accesses in database organizations. *IEEE Transactions on Knowledge and Data Engineering*, 6(3), 1994.
7. A. Eickler, C. A. Gerlhof, and D. Kossmann. Performance evaluation of OID mapping techniques. In *Proceedings of the 21st VLDB Conference*, 1995.
8. G. Gardarin, J.-R. Gruser, and Z.-H. Tang. A cost model for clustered object-oriented databases. In *Proceedings of the 21st VLDB Conference*, 1995.
9. W. S. Luk. On estimating block accesses in database organizations. *Communications of the ACM*, 26(11), 1983.
10. L. F. Mackert and G. M. Lohman. Index scans using a finite LRU buffer: A validated I/O model. *ACM Transactions on Database Systems*, 14(3), 1989.
11. M. L. McAuliffe. *Storage Management Methods for Object Database Systems.* PhD thesis, University of Wisconsin-Madison, 1997.
12. K. Nørvåg and K. Bratbergsengen. An analytical study of object identifier indexing. Technical Report IDI 4/98, Norwegian University of Science and Technology, 1998.
13. K.-Y. Whang and G. Wiederhold. Estimating block accesses in database organizations: A closed noniterative formula. *Communications of the ACM*, 26(11), 1983.
14. S. B. Yao. Approximating the number of accesses in database organizations. *Communications of the ACM*, 20(4), 1977.

Appendix D

Optimizing OID Indexing Cost in Temporal Object-Oriented Database Systems

This appendix contains the paper presented at the 5th International Conference on Foundations of Data Organization (FODO'98), held in Kobe, Japan, November 1998.

Optimizing OID Indexing Cost in Temporal Object-Oriented Database Systems

Kjetil Nørvåg and Kjell Bratbergsengen

Department of Computer and Information Science
Norwegian University of Science and Technology, Norway

{*noervaag,kjellb*}*@idi.ntnu.no*

Abstract

In object-oriented database systems (OODB) with logical OIDs, an OID index (OIDX) is needed to map from OID to the physical location of the object. In a transaction time temporal OODB, the OIDX should also index the object versions. In this case, the index entries, which we call object descriptors *(OD), also include the commit timestamp of the transaction that created the object version. In this paper, we develop an analytical model for OIDX access costs in temporal OODBs. The model includes the index page buffer as well as an OD cache. We use this model to study access cost and optimal use of memory for index page buffer and OD cache, with different access patterns. The cost models in this paper can be of valuable use for optimizers and automatic tuning tools in temporal OODBs. The primary context of this paper is OID indexing in a temporal OODB, but the results are also relevant in the context of general secondary index access cost and index entry caching.*

Keywords Index organization, temporal database systems, object-oriented database systems

1 Introduction

In a traditional object-oriented database system (OODB), updating an object makes the old version unaccessible. In a temporal object-oriented database system (TOODB), on the other hand, updating an object simply creates a new version of the object, the old version is still accessible. In a transaction time TOODB, which is the context of this paper, a system maintained timestamp is associated with every object version. This timestamp is the commit time of the transaction that created this version of the object.[1]

In an OODB, an object is uniquely identified by an object identifier (OID). The OID can be physical, which means that the disk page of the object is given directly from the OID, or logical, which means that an OID index (OIDX) is needed to map from logical OID to the physical location of the object. A physical OID scheme is inflexible, and in a temporal OODB, logical OIDs is the only reasonable alternative, because of objects being moved. The entries in the OIDX, the *object descriptors* (OD)s, contain administrative information, including information to do the mapping from logical OID to physical address. An object is accessed via its OID, hence, lookup in the OIDX have to be as efficient as possible. The OIDX can be quite large, typical in the order of 20% of the size of the database itself [4]. This means that in general, only a small part of the OID index fits in main memory, and that OID index retrieval can become a bottleneck if efficient access and buffering strategies are not applied. An important difference between OIDX management in non-temporal and temporal OODBs, is that with only one version of an object (non-temporal), the OIDX needs only to be updated when an object is created, which can be done efficiently as an efficient append-only operation. In a TOODB however, the OIDX must be updated every time an object is updated. An object update creates a new object version, without deleting the previous version, hence, a new OD for the new version have to be inserted into the OIDX. The index pages will in general have low locality (the unique part of an OID is usually an integer that will always be assigned monotonic increasing values), and index updates might become a serious bottleneck in a TOODB.

We will in this paper study the cost of OIDX accesses, inserts as well as lookups, and the use of buffering to speed up the OID mapping. Our approach is based on analytical modeling. Compared to simulations, another common used approach, analytical modeling has the advantage that it is easy to change parameters, to be able to explore larger areas of the parameter space. With an analytical model, cost and optimal values for configurable parameters can be obtained fast. This is important for automatic system tuning and query planning. As systems get larger, more complex, and more flexible, automatic tuning is needed, or at least support to a greater extent than today. One way to accomplish this, is to use analytical system models to compute optimal configuration parameters, e.g., buffer sizes, so that the systems can adapt to changing work-

[1] In the other common category of temporal database systems, valid time database system, a time interval is associated with every object, denoting the time interval which the object is valid in the modeled world.

The 5th International Conference on Foundations of Data Organization (FODO'98), Kobe, Japan, November 1998.

loads, and always give as high performance as possible. Cost models are also useful for query planning, to choose the best query plans, and the results from this paper can also be used for system design. It should also be noted that even though our primary context for this paper is OID indexing, the results are also relevant to entry access cost and entry caching for general secondary indexes.

The organization of the rest of the paper is as follows. In Section 2 we give an overview of related work. In Section 3 we describe object and index management in TOODBs. In Section 4 we describe our index entry access model. In Section 5 we describe briefly the Bhide, Dan and Dias LRU buffer model [2], and our hierarchical index buffer model. In Section 6 we develop an the OID access cost model, and in Section 7 we use this cost model to study how different memory sizes, index sizes, access patterns and disk page sizes affect the performance. Finally, in Section 8, we conclude the paper and outline issues for further research.

2 Related Work

Several cost models for OODBs have been developed, for example [1, 6], but they do not include buffer and OID mapping cost. In a previous paper, we have studied index lookup cost and optimal use of memory in a non-temporal, one-version, OODB [13]. The buffer and tree models have been compared with simulation results. Detailed results from the simulations with different index sizes, buffer sizes, index page fanout, and access patterns can be found in [14].

Temporal database systems are in general still an immature technology, and in the case of transaction time TOODBs, we are only aware of one prototype[2] that have temporal OID indexing [15]. The work described in this paper is part of the Vagabond TOODB project [11, 12].

3 TOODB Object and Index Management

We start with a description of how OID indexing and version management can be done in a TOODB. This brief outline is not based on any existing system, but the design is close enough to current OODBs to make it possible to integrate if desired, and it will also be used as a basis for the OID indexing in the Vagabond TOODB.

3.1 Temporal OID Indexing

In a traditional OODB, the OIDX is usually realized as a hash file or a B-tree, with ODs as entries, and using the OID as the key. In a TOODB, we have more than one version of some of the objects, and we need to be able to access current as well as old versions efficiently. If access is mostly reading current objects, it is efficient to have two indexes, one with ODs representing the current version of the objects, and one with ODs representing historical objects (i.e, previous versions). The problem with this approach, is that every time a new version is created, we have to update *two* indexes. A second approach, is to use a linked list of versions for each object. If accesses are mostly of the type "get all versions of an object with OID i", an efficient alternative is to use a linked list of versions for each object. However, access to a particular version, valid at time t, is very costly with this approach, because we have to traverse the object chain.

Our approach to indexing is to have *one* index structure, containing all ODs, current as well as previous versions. While several efficient multiversion access methods exist, e.g., TSB-tree [7] and LHAM [10], they are not suitable for our purpose. We will never have search for a (consecutive) range of OIDs, OID search will always be for *perfect match*, and most of them are assumed to be to the current version. TSB-trees provides more flexibility than needed, e.g., combined key range and time range search, which implies an extra cost, while LHAM can have a high lookup cost when the current version is to be searched for.

In this paper, we assume one OD for each object version, stored in a B^+-tree. We include the commit time $TIME$ in the OD, and use the concatenation of OID and time, $OID\|TIME$, as the index key. By doing this, ODs for a particular OID will be clustered together in the leaf nodes, sorted on commit time. As a result, search for the current version of a particular OID as well as retrieval of a particular time interval for an OID can be done efficiently.

When a new object is *created*, i.e., a new OID allocated, its OD is appended to the index tree as is done in the case of the Monotonic B^+-tree [5]. This operation is very efficient. However, when an object is *updated*, the OD for the new version *have to be inserted into the tree.*

It should be noted that this OIDX is inefficient for many typical temporal queries. As a result, additional secondary indexes can be needed, of which both TSB-tree and LHAM are good candidates. However, *the OIDX is still needed*, to support navigational queries.

3.2 Temporal Object Management

In a one-version (non-temporal) OODB, space is allocated for an object when it is created, and updates to the objects are done in-place. This implies that after an object update, the previous version of the object is not available. The physical location of the new version is the same as the previous version, hence, the OIDX needs only to be updated when objects are created and when they are deleted.

In a TOODB, it is usually assumed that most accesses will be to the current versions of the objects in the database. To keep these accesses as efficient as pos-

[2] Support for versioning exists in most OODBs, but not temporal management, indexing, and operations.

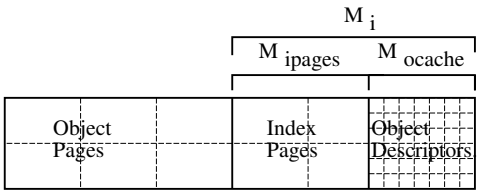


Figure 1: Object page buffer, index page buffer, and OD cache in a page server TOODB.

sible, and benefit from object clustering,[3] the database is partitioned, with current objects in one partition, and the previous versions in the other partition, in the *historical database*. When an object is updated in a TOODB, the previous version is first moved to the historical database, before the new version is stored in-place in the current database. The OIDX needs to be updated *every time an object is updated*. As long as the modified ODs are written to the log before commit, we do not need to update the OIDX itself immediately. This is done in the background, and can be postponed until the second checkpoint after the OD have been written to the log. Index pages will be written to disk either because of checkpointing, or because of buffer replacement.

Not all the data in a TOODB is temporal, for some of the objects, we are only interested in the current version. To improve efficiency, the system can be made aware of this. In this way, some of the data can be defined as non-temporal. Old versions of these are not kept, and objects can be updated in-place as in a one-version OODB, and the costly OIDX update is not needed when the object is modified. This is an important point, using an OODB which efficiently supports temporal storage, should not reduce the performance of applications that do not utilize this feature.

3.3 Index Page Buffer and OD Cache

To reduce disk I/O, the most recently used *index pages* are kept in an *index page buffer*. OIDX pages will in general have low locality, and to increase the probability of finding a certain OD needed for a mapping from OID to physical address, it is also possible to keep the most recently used *index entries* (the ODs) in a separate OD cache, as is done in the Shore OODB [9]. With low locality on index pages, a separate OD cache utilizes memory better, space is not wasted on large pages where only small parts of them will be used. The size of the OD cache can be fixed (set at system startup time), or adaptive. The buffer in a page server OODB is illustrated on Figure 1. In the rest of this paper, we will denote the index page buffer size as M_{ipages}, the OD cache size as M_{ocache}, and the sum of these as M_i, as illustrated on the figure.

[3] It is also possible that in a TOODB application, a good object clustering includes historical objects as well as current objects. This should be studied further, but does not have any implications to the results studied here, all updates to objects will necessarily necessitate allocations of space for the new object, and an OIDX update.

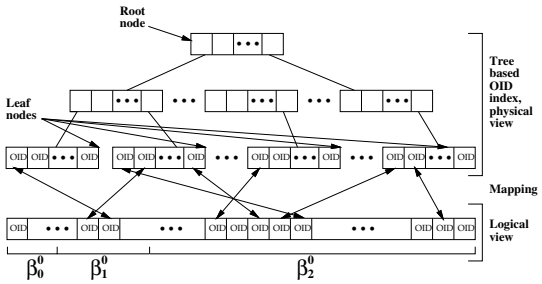


Figure 2: OID index. The lower part shows the index from a logical view, the upper part is as an index tree, which is how it is realized physically. We have indicated with arrays how the entries are distributed over the leaf nodes.

4 Index Entry Access Model

We assume accesses to objects in the database system to be random, but skewed (some objects are more often accessed than others). We further assume it is possible to (logically) partition the range of OIDs into partitions, where each partitions has a certain size and access probability. This is illustrated on the bottom of Figure 2 (Note that this is not how it is stored on disk, this is just a model of accesses). We consider a database in a stable condition, with a total of N_{objver} objects versions (and hence, N_{objver} index entries). Note that with the OIDX described in Section 3.1, performance is not dependent of the number of existing versions of an object, only the total number of versions in the database.

In many analysis and simulations, the 80/20 model is applied, where 80% of the accesses go to 20% of the database. While this is satisfactory for analysis of some problems, it has a major shortcoming when used to estimate the number of distinct objects to be accessed. When applied, it gives a much higher number of distinct accessed objects than in a real system. The reason is that for most applications, inside the hot spot area (20% in this case), there is an even hotter and smaller area, with a much higher access probability. This is even more important for a temporal database. Most of the accesses will be to a small number of the current versions. With a large number of previous versions, this hot spot area will be much smaller and "hotter" than the one in a typical "traditional" database. This has to be reflected in the model.

5 Buffer Performance Model

The buffer performance model used in this study is presented in detail in [13]. Due to space constraints, we only present the notations in this paper.

The BDD LRU Buffer Model. In our analysis, we need to estimate the buffer hit probability in an LRU managed buffer. We do this with the Bhide, Dan and Dias LRU buffer model (BDD) [2]. A database in the BDD model has a size of N data granules (pages or objects), partitioned into p partitions. Each partition contains β_i of the data granules, and α_i of the accesses are done to each partition. The distributions *within* each of the partitions are assumed to be uniform. All accesses are assumed to be independent. We denote a particular partitioning as Π.

The steady state average buffer hit probability is denoted $P_{\text{buf}}(B, N, \Pi)$, where B is the number of data granules that fits in the buffer. We also use the BDD model to calculate the total number of distinct data granules accessed after n accesses to the database, $N_{distinct}(n, N, \Pi)$.

Index Page Access Model. As noted, we can assume low locality in index pages. Because of the way OIDs are generated, entries from a certain partition are not clustered in the index. This is illustrated in Figure 2, where a leaf node containing index entries contains unrelated entries from different partitions. This means that the access pattern for the leaf nodes is different from the access pattern to the database from a logical view.

We use the initial index entry partitioning (the index entry access pattern) as basis for deriving the index page partitioning (the index page access pattern). The index page partitioning is a function of the initial index entry partitioning. We will in the following use Π_A as short for the data entry access partitioning, and Π_L as short for the leaf node access partitioning. Π_L is calculated from Π_A [14].

General Index Buffer Model. The BDD LRU buffer model only models independent, non-hierarchical, access. Modeling buffer for hierarchical access is more complicated. Even though searches to the leaf page can be considered to be random and independent, nodes accessed during traversal of the tree are *not* independent. We have extended the original model to be able to analyze buffer performance in the case of hierarchical index accesses as well [14]. We denote the overall buffer probability as $P_{\text{buf_ipage}}(B, N_{\text{tree}})$, where N_{tree} is the number of index nodes in the OIDX.

6 Index Access Cost

Analytical modeling in database research has mostly focused on I/O costs. This is the most significant cost factor, and in reasonable implementations, the CPU processing should go in parallel with I/O transfer making the CPU cost "invisible". With increasing amounts of main memory available, this is not necessarily correct, but CPU costs can easily be incorporated into analytic models, and hence we consider it as an orthogonal issue to the one discussed in this paper (though it should be noted, that CPU cost should not affect the qualitative results in this paper). A more important aspect of the increasing amount of main memory, however, is that buffer characteristics become more important, hence, the increased buffer space available must be reflected in the models.

We use a traditional disk model, where the cost of reading a block from disk is the sum of the start up cost T_{start} and the transfer cost $T_{transfer}$. In our model, the average start up cost is fixed, and is set equivalent to t_r, the time it takes to do one disk revolution. The transfer cost is directly proportional to the block size, and is equivalent to reading disk tracks contiguously, e.g., transfer cost is equal to $\frac{b}{V_s}t_r$, where b is the block size to be transferred, and V_s is the amount of data on one track. Thus, the total time it takes to transfer one block is $T_b(b) = T_{start} + T_{transfer} = t_r + \frac{b}{V_s}t_r$.

The time to read or write a random index page is $T_P = T_b(S_P)$, where S_P is the index page size. In this paper, we do not consider the cost of reading and writing the objects themselves, or log operations. Those costs are independent of the indexing costs, usually done on separate disks, and are issues orthogonal to the ones studied in this paper.

6.1 Index Tree Size

If we assume an index tree with index node size of S_P, index entry size of S_{ie}, space utilization U (typically 0.69 for a B-tree), and N_{objver} object versions to be indexed, the fanout $F = \lfloor US_P/S_{\text{ie}} \rfloor$, the number of nodes at each level in the tree is $N^i_{tree} = \lceil \frac{N^{i-1}_{\text{tree}}}{F} \rceil$ and the number of leaf nodes is $N^0_{\text{tree}} \approx \frac{N_{\text{objver}}}{U\lfloor S_P/S_{\text{ie}} \rfloor}$. The total number of nodes in the tree is $N_{\text{tree}} = \sum_{i=0}^{H-1} N^i_{\text{tree}}$, where $H = 1 + \lceil \log_F N^0_{\text{tree}} \rceil$ is the number of levels in the tree.

6.2 Buffer Hit Probabilities

A certain amount of memory, M_{ipages}, is reserved for the index page buffer, and M_{ocache}, is reserved for the OD cache. If we assume the size of each OD is S_{od}, and an overhead of S_{oh} bytes is needed for each entry in the OD cache, the number of entries that fits in the OD cache is approximately $N_{\text{ocache}} \approx \frac{M_{\text{ocache}}}{(S_{\text{od}}+S_{\text{oh}})}$. Accesses to the OD cache can be assumed to follow the assumptions behind the BDD model, they are independent random requests, and by applying this model with object entries as data granules the probability of an OD cache hit is $P_{\text{ocache}} = P_{\text{buf}}(N_{\text{ocache}}, N_{\text{objver}}, \Pi_A)$.

Accesses to the index page buffer, on the other hand, follows the general index buffer model (Section 5). The number of index pages that fits in the buffer is approximately $N_{\text{ibuf}} \approx \frac{M_{\text{ipages}}}{(S_{\text{P}}+S_{\text{oh}})}$, and the probability of an index page buffer hit is $P_{\text{buf_ipage}}(N_{\text{ibuf}}, N_{\text{tree}})$.

The results in the following analysis are highly dependent of the amount of overhead S_{oh} needed for each entry. Of course, with a minimal overhead, it would always be beneficial to use as much as possible of the total index memory as an OE cache. The most reasonable way to implement it, and at the same time keep the

CPU cost low, is to use an hash table to provide fast access to the entries. In that case, approximately two pointers are needed for each entry on average. We also need some additional data structures to do the buffer management. Even though we base the analysis on an LRU buffer, a clock algorithm will probably be used. It has performance close to LRU [3], but has less storage overhead, only one bit is needed for each entry. This is small enough to ignore in this analysis.

6.3 Index Lookup Cost

With a probability of P_{ocache}, the OID entry requested is already in the OD cache, but for $(1 - P_{\text{ocache}})$ of the requests, we have to access the index pages, and one or more disk accesses might be needed. The probability of a given index page being in memory is *on average* $P_{\text{buf_ipage}}$. To access an entry (OD) in an index page, it is necessary to traverse the H levels from the root to a leaf page. To do this, we need $(1 - P_{\text{buf_ipage}})H$ disk accesses. The average cost of an index lookup is $T_{\text{lookup}} = (1 - P_{\text{ocache}})(1 - P_{\text{buf_ipage}})HT_P$.

6.4 Index Update Cost

We do not need to update the index pages in the OIDX immediately. This is done in the background, and can be postponed, increasing the probability that several updates can be done to the index page before it is written back. We calculate the average index update cost as the total index update cost during one checkpoint interval, divided on the number of index updates. In this context, we define the checkpoint interval to be the number of objects that can be written between two checkpoints. The number of written objects, N_{CP}, includes created as well as updated objects. $P_{new}N_{CP}$ of the written objects are creations of new objects, and $(1-P_{new})N_{CP}$ of the written objects are updates of existing objects. We assume that memory is large enough to keep all dirty ODs through one checkpoint interval, and that delete and compacting pages can be done in background.

Creation of new object descriptors. New object descriptors are created when new objects are created. The number of created objects is $N_{CR} = P_{new}N_{CP}$. When new objects are created, their ODs are appended to the index (we do not distribute the entries over the old node and the new when the rightmost nodes are split), and we have clustered updates. This contributes to $N_n^0 = \frac{N_{CR}}{\lfloor S_P/S_{\text{od}} \rfloor}$ created leaf pages. This is a subtree in the index tree, of height H_s, with $S_n = \sum_{i=0}^{H_s-1} N_n^i$ pages. The total cost of creating these object descriptors is the cost of writing S_n index pages to the disk, no installation read is needed for these pages. Assuming that the disk is not too fragmented, these pages can be written in one operation, most of them sequentially:

$$T_{writenew} = T_b(S_n S_P)$$

Modification of existing object descriptors. When an object is updated, a new object version is created, and a new OD has to be inserted into the OIDX. The number of updated objects is $N_U = N_{CP} - N_{CR}$. Updating the index involves a page installation read, where the page where the last (current) version resides is read from disk, if the page is not already in the buffer. The cost of this is T_{lookup} for each *distinct* object modified. The number of distinct updated objects is:

$$N_{DU} = N_{distinct}(N_{CP} - N_{CR}, N_{\text{objver}}, \Pi_A)$$

However, as noted in Section 3.2, not all objects in a TOODB are temporal. We denote the fraction of the data accesses going to temporal objects as $P_{temporal}$. Only updates of these objects alter the OIDX, updates of non-temporal objects will done in-place. The number of distinct updated temporal objects is:

$$N_{DU}^V = P_{temporal}N_{DU}$$

The number of leaf pages to be accessed as a part of the installation read:

$$N_m = N_{distinct}(N_{DU}^V, N_{tree}^0, \Pi_L)$$

If there is space for the new OD in the leaf node, it can be inserted there, and the node can be written back. If there is no space in the node, the node is split, a process done recursively, possibly to the root. If a node is split, the parent node has to be updated as well (except in the case when the root node is split, in this case, the height of the tree is increased with one level). Because of the possibility of page splits, determining the update cost is difficult. With sufficiently many entries on each index node, the probability of page split is small enough to be neglected [16]. However, for some pages, there are more than one insertion to that page (possibly generated by several updates to one object during one checkpoint interval, remember that *each update creates a new entry to be inserted into the OIDX*). Thus, we include the page split in our cost functions. According to Loomis [8], the probability of a split in a B-tree of order m is less than $\frac{1}{\lceil m/2 \rceil - 1}$, so we approximate $P_{split} \approx \frac{1}{\lceil (S_P/S_{\text{od}})/2 \rceil - 1}$. For each split, the new page needs to be written back, as well as the updated parent node. However, note that there may be several splits affecting one parent node, in this case, it needs only be written back once. The resulting total write back cost is:

$$\begin{aligned} T_{writeback} &= (N_m + N_m 2P_{split})T_P \\ &= N_m(1 + 2P_{split})T_P \end{aligned}$$

If the checkpoint interval is sufficiently large, it is more efficient to read the complete index, update the index nodes, and write it back (if memory is not large enough, this is done in segments). This will be very efficient, as the reading and writing will be sequential. The cost of this is:

$$T_{readwrite_all} = 2T_b(N_{tree}S_P)$$

The total cost of index update during one checkpoint interval:

$$T_{update_total} = T_{writenew} + \min(N_m T_{lookup} + T_{writeback}, T_{readwrite_all})$$

The average index update cost per object:

$$T_{update} = \frac{T_{update_total}}{N_{CP}}$$

7 Analytical Study

We have now derived the cost functions necessary to calculate the average cost of OIDX access under different system parameters and access patterns. We will in the following sections study in detail how different values for these parameters affects the access cost, and how the optimal OD cache size changes as well. The mix of updates and lookups to the OIDX affects the optimal parameter values, and they should be studied together. If we denote the probability that an object operation is a write, as P_{write}, the average index access cost is the average of the cost of all index lookup and index update operations:

$$T_{access} = (1 - P_{write})T_{lookup} + P_{write}T_{update}$$

We define optimal OD cache size as the size of the OD cache that, given a certain amount of memory available for index pages and OD cache,[4] gives the lowest average index access cost, i.e., find the M_{ocache} that minimizes $T_{acccess}$, given the invariant:

$$M_i = M_{\text{ocache}} + M_{\text{ipages}} = Constant$$

In the rest of this paper, we give OD cache size as a fraction of the total index memory size. The index memory size itself is given as a fraction of the total space required for the whole index tree, i.e., when $\frac{\text{Index Buffer Size}}{\text{Total Index Size}} = 1.0$, all the pages in the index tree fits in memory.

In this study, we have chosen five access patterns, the partition sizes and access probabilities are summarized in Table 1 (note that this is the *OID access pattern*, and not the index page access pattern). We call each of these patterns a *partitioning set*. β_i^0 denotes the size of partition i, as a fraction of the total database size, and α_i^0 denotes the fraction of the accesses done to partition i.

In the two first partitioning sets, we have three partitions, extensions of the 80/20 model, but with the 20% hot spot partition further divided. In the first partitioning set, we have a 1% hot spot area, in the other, a 0.1% hot spot area. The three other sets have each two partitions, with hot spot areas of 5%, 10%, and 30%.

[4] Note that memory needed for the buffering of data pages/objects is not included, this issue is orthogonal to the one studied here.

Set	β_0^0	β_1^0	β_2^0	α_0^0	α_1^0	α_2^0
3P1	0.01	0.19	0.80	0.64	0.16	0.20
3P2	0.001	0.199	0.80	0.64	0.16	0.20
2P7030	0.30	0.70	-	0.70	0.30	-
2P9010	0.10	0.90	-	0.90	0.10	-
2P9505	0.05	0.95	-	0.95	0.05	-

Table 1: Partition sizes and partition access probabilities for the partitioning sets used in this study.

The analysis have been done with a database with $N_{\text{objver}} = 20$ million objects. Unless otherwise noted, results and numbers in the next sections are based on calculations with an OD size of 32 bytes, index page size of 8 KB, access pattern according to partitioning set 1, write probability $P_{write} = 0.2$, object create probability $P_{new} = 0.2$, and all objects in the database temporal. Checkpoint interval is $N_{CP} = 20000$ objects. The database size and the checkpoint interval is smaller than those used in most real world applications, but the qualitatively results and fractions stay the same when these numbers are scaled. The same applies to the number of disks. Performance can be increased by partitioning the index over N_{disks} disks. In this case, $T_b(b) = (t_r + \frac{b}{V_s}t_r)/N_{disks}$.

Note that even though some of the parameter combinations in the following sections are unlikely to represent the average over time, they can occur in periods, e.g., more write than read operations. It is in situations like this that adaptive self tuning systems are most important, when parameter sets differs from the average, which systems traditionally have been tuned against.

7.1 Optimal OD Cache Size

Figure 3 shows how optimal OD cache size changes with different access patterns. On the left hand side, we have the optimal OD cache size for the partitioning sets with two partitions. We see clearly how the optimal OD cache size fraction reaches the top at 5% and 10% for the partitioning sets 2P9505 and 2P9010, where the hot spot area is in the OD buffer. When the hot spot area gets even wider, with partitioning set 2P7030, the hit probability for an index page when doing installation read is so small that the memory is better spent caching ODs, to reduce the lookup cost.

For partitioning sets 3P1 and 3P2, to the right on Figure 3, we see the same. With a significantly small and frequently accessed hot spot area, it is important to get the hottest ODs in the cache. When the cache is large enough to fit these data, the rest of the index memory is better utilized as index page buffer. With a wider hot spot area, as in 3P1, the same phenomenon as for the 2P7030 set happens.

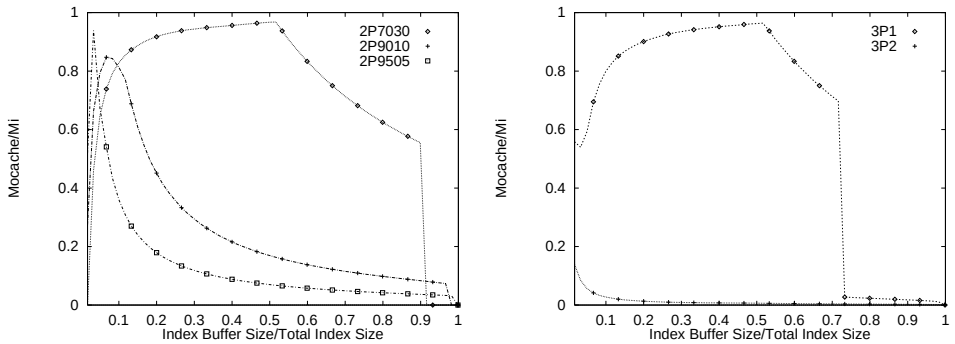


Figure 3: Optimal OD cache size with different partitioning sets.

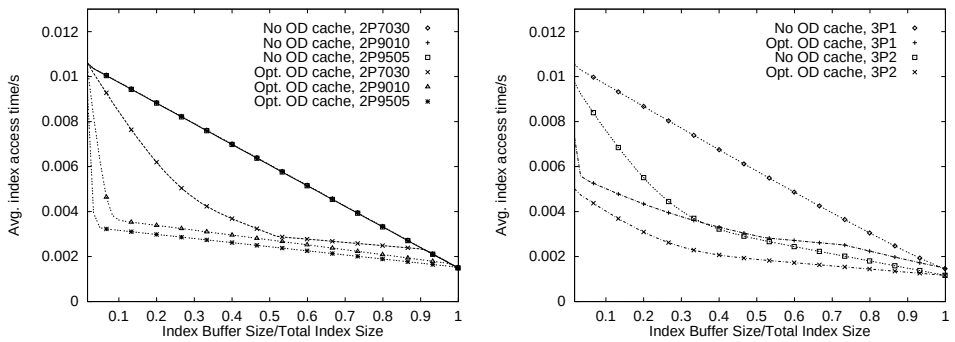


Figure 4: Access cost.

7.2 Access Cost

Figure 4 illustrates how the access cost decreases with increasing amounts of available index memory. For each of the partitioning sets, the figure show the access cost without OD cache, and the access cost with an OD cache of optimal size.

From the figure, we see that even when the whole index fits in memory, the access cost is quite large. This might come as a surprise, considered that with the whole index in memory, we avoid the costly installation reads. What happens, is that a small percentage of the updates creates page splits. Even though the new pages can be written efficiently in one operation, their parent nodes have to be updated. This update is done in-place, and is a costly random write.

Interesting to note, is the access cost in the case of using the partition sets with only two partitions. In this case, if an OD cache is not employed, all accesses have to be done to the index pages. When the hottest hot spot is not small enough to make certain index pages become hot spots, the accesses to the index pages are close to uniform. As a result, the access cost is the same for all the 2 partition partitioning sets. When an OD cache is employed, we see clearly how performance increases for the access patterns with the most narrow and frequently accessed hot spots.

Figure 4 illustrates well the gain from employing an optimal OD cache size, compared to no OD cache at all. The gain is highest when we have a small hot spot area. The gain increases with increasing index memory size, up to the point where the whole hot spot area fits in the OD cache.

7.3 Effect of Update Ratio

The ratio of object read versus object write is important. On Figure 5 we see how changing P_{write} affects the optimal OD cache size as well as the access cost. The object create probability is held constant at $P_{new} = 0.2$.

When the update rate is relatively low, it is beneficial to use much of the memory to cache ODs. However, as the write ratio increases, the cached ODs will be less useful, because the whole index page is needed when an entry is to be updated. From the figure we see that the memory allocation strategy changes when P_{new} is between 0.4 and 0.5.

To the right on the figure, we see how the cost with different write ratios. Updating the index is costly, as is evident from the figure.

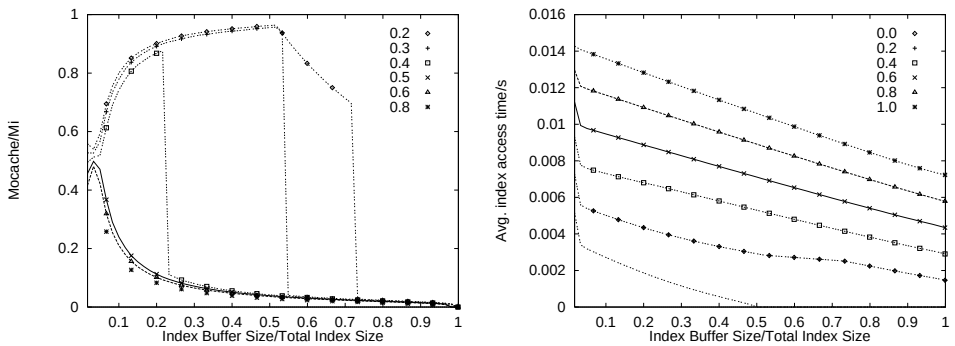


Figure 5: Optimal OD cache size with different values of P_{write} to the left, access cost with different values of P_{write} to the right.

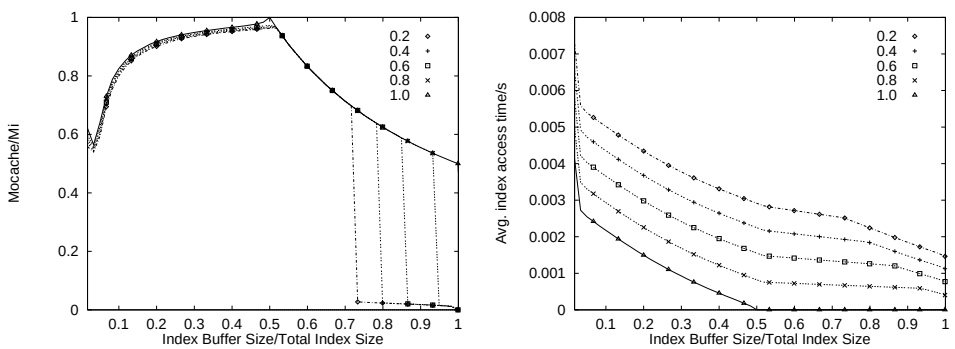


Figure 6: Optimal OD cache size with different values of P_{new} to the left, access cost with different values of P_{new} to the right.

7.4 Effect of Object Creation Ratio

On Figure 6 we see how changing P_{new} affects the cost as well as the optimal OD cache size. The write probability is held constant at $P_{write} = 0.2$. The difference in optimal cache size for different values of P_{new} is only marginal, except for large memory sizes, where we have a sharp drop. The drop comes first for the low values of P_{new}. The reason is that updates needs index pages from the index as a part of the installation read. With large values of P_{new}, this is not as important.

To the left on Figure 6 we see again how important the cost of object index update is, and how the mix of create and update affects this. When objects written are mostly new objects, this can be done very efficiently. Appending ODs to the index is inexpensive.

7.5 Effect of Checkpoint Interval Length

Increasing the checkpoint interval length can reduce the access cost, illustrated to the right on Figure 7. The cost we have to pay for this, is a longer recovery time after a crash, as a larger amount of log have to be processed.

We have studied how different lengths of the checkpoint interval affects the optimal M_{ocache}. Figure 7 shows the optimal OD cache size for different values for N_{CP}. An important observation is that with other parameters held constant, different values for N_{CP} give exactly one of two different values for the optimal OD cache size. This can be seen from Figure 7. Only four of the values are shown on the figure, but the result is the same for other values a well.

7.6 The Effect of Non-Temporal Objects

We have, until now, assumed that all objects in the database are temporal. Figure 8 illustrates the effects of non-temporal objects on the optimal OD cache size and the access cost. The optimal OD cache size is independent of the fraction of temporal objects.

The cost is highly dependent of the fraction of temporal objects. As emphasized earlier in this paper, maintaining versions is very costly. A low fraction of temporal objects implies less index updates. The extreme is $P_{temporal} = 0.0$, which is a non-temporal database, a traditional OODB.

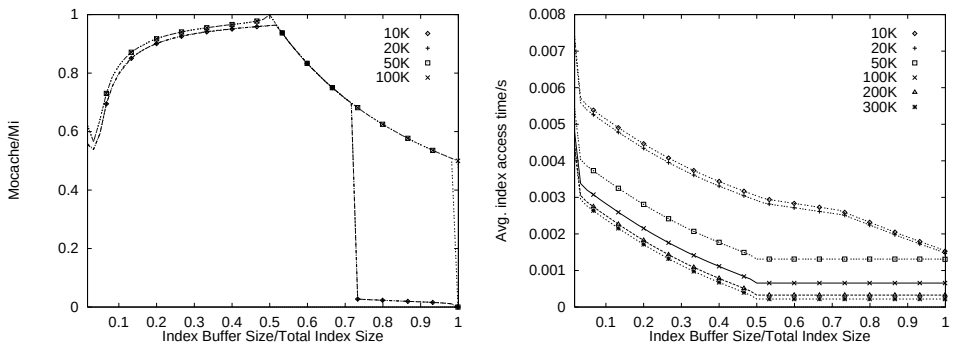


Figure 7: Optimal OD cache size with different values of N_{CP} to the left, access cost with different values of N_{CP} to the right.

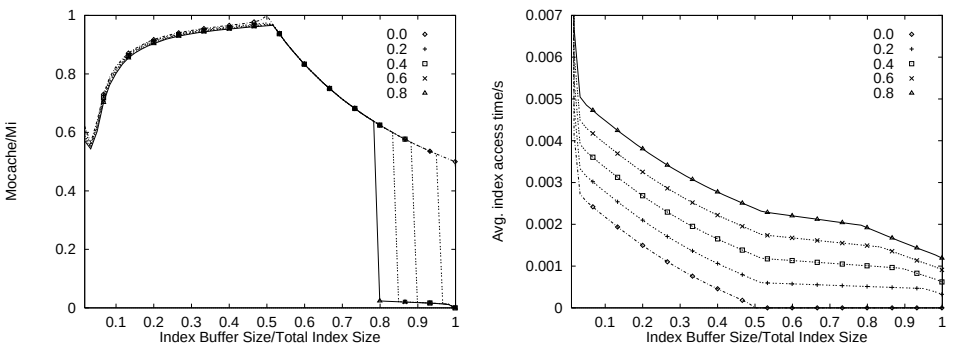


Figure 8: Optimal OD cache size with different values of $P_{temporal}$ to the left, access cost with different values of $P_{temporal}$ to the right.

Note the similarities between Figure 6 and Figure 8. The reason is that, increasing the create probability and reducing the versioning, each increases the number of index appends relative to index inserts.

7.7 Page Size

The optimal page size is a compromise of two contradicting factors. Because of low locality, large page sizes in an OIDX means more wasted space in the index page buffer, and the optimal page size is thus much smaller. However, small pages also results in more levels in the tree. Even though in most cases upper levels of the index tree will be resident in memory, a tree with smaller page size also needs more space, reducing the buffer hit probability. We can see that there are two strategies for efficiency: Either large pager, which is particularly advantageous for the creation of objects, and small pages, to capture the fact that there is low level of sharing. Our analysis shows that the page size giving the lowest cost over most of the area, is 2 KB. This is less than the 8 KB blocks commonly used, and this result might come as a surprise, as relational database systems have started to employ larger page sizes on their index structures. However, the index access pattern in a typical relational database systems is different from the index modeled in this paper. In a relational database system, index scan is very common, which benefits from large page sizes.

Interesting to note is also how the optimal OD cache size changes with page size, illustrated on Figure 9. The reason for this, is that with large pages, we will in general benefit from more OD caching, because only a small part of the pages will be accessed. This outweights the disadvantages of the installation read needed when the page is updated.

8 Conclusions and Future Work

We have in this paper developed an analytical model for OIDX access cost in a transaction time temporal database system. We have used this model to study the behavior of the OIDX under different access patterns. The results show that:

1. The OIDX access cost can be high, and can easy become a bottleneck in large TOODBs.

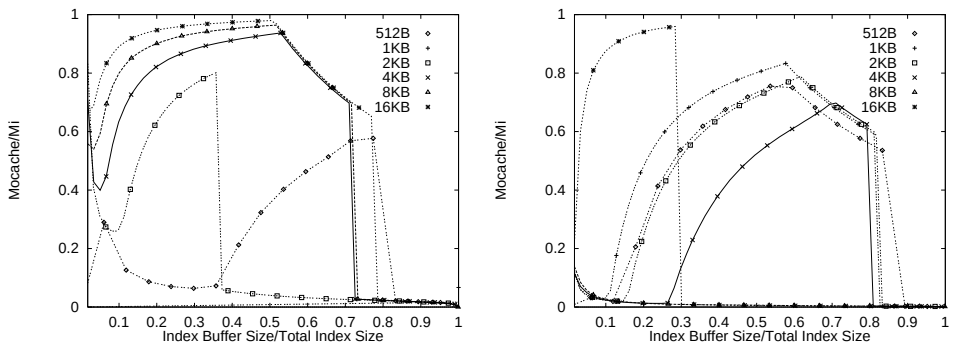


Figure 9: Optimal OD cache size with different page sizes, partitioning set 3P1 to the left, 3P2 to the right.

2. The optimal OD cache size can be large, and the gain from using an optimal size is considerable. Having an optimally tuned system is important.

3. Access pattern in a database system is very dynamic, and the system should be able to detect this, and tune the size of index page buffer and OD cache size accordingly. The cost models in this paper can be of valuable use for optimizers and automatic tuning tools in temporal OODBs.

As is obvious from the results in the previous sections, OID index maintenance will be costly. To get an acceptable performance, most of the OIDX must be in memory to avoid the costly installation reads of index pages. The cost can be reduced by partitioning the index over several disks, and it is possible that the object indexing itself can be optimized, this should be studied further.

The model and results in this paper could also be used in the context of caching general secondary index entries in relational as well as object-oriented database systems. This issue is independent of the one studied here, but caching entries in secondary indexes should be an interesting further work.

References

[1] E. Bertino and P. Foscoli. On modeling cost functions for object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3), 1997.

[2] A. K. Bhide, A. Dan, and D. M. Dias. A simple analysis of the LRU buffer policy and its relationship to buffer warm-up transient. In *Proceedings of the Ninth International Conference on Data Engineering*, 1993.

[3] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4), 1984.

[4] A. Eickler, C. A. Gerlhof, and D. Kossmann. Performance evaluation of OID mapping techniques. In *Proceedings of the 21st VLDB Conference*, 1995.

[5] R. Elmasri, G. T. J. Wuu, and V. Kouramajian. The time index and the monotonic B^+-tree. In A. U. Tansel, J. Clifford, S. K. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal databases: theory, design and implementation*. The Benjamin/Cummings Publishing Company, Inc., 1993.

[6] G. Gardarin, J.-R. Gruser, and Z.-H. Tang. A cost model for clustered object-oriented databases. In *Proceedings of the 21st VLDB Conference*, 1995.

[7] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD*, 1989.

[8] M. E. Loomis. *Data Management and File Structures*. Prentice Hall, 1989.

[9] M. L. McAuliffe. *Storage Management Methods for Object Database Systems*. PhD thesis, University of Wisconsin-Madison, 1997.

[10] P. Muth, P. O'Neil, A. Pick, and G. Weikum. Design, implementation, and performance of the LHAM log-structured history data access method. In *Proceedings of the 24th VLDB Conference*, 1998.

[11] K. Nørvåg and K. Bratbergsengen. Log-only temporal object storage. In *Proceedings of the 8th International Workshop on Database and Expert Systems Applications, DEXA 97*, 1997.

[12] K. Nørvåg and K. Bratbergsengen. Write optimized object-oriented database systems. In *Proceedings of the XVII International Conference of the Chilean Computer Science Society, SCCC'97*, 1997.

[13] K. Nørvåg and K. Bratbergsengen. An analytical study of object identifier indexing. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications*, 1998.

[14] K. Nørvåg and K. Bratbergsengen. An analytical study of object identifier indexing. Technical Report IDI 4/98, Norwegian University of Science and Technology, 1998. Available from http://www.idi.ntnu.no/grupper/DB-grp/.

[15] T. Suzuki and H. Kitagawa. Development and performance analysis of a temporal persistent object store POST/C++. In *Proceedings of the 7th Australasian Database Conference*, 1996.

[16] J. D. Ullman. *Principles of database and knowledge-base systems*. Computer Science Press, 1988.

Appendix E

The Persistent Cache: Improving OID Indexing in Temporal Object-Oriented Database Systems

This appendix contains the paper presented at the 25th International Conference on Very Large Databases (VLDB'99), Edinburgh, Scotland, UK, September 1999.

The Persistent Cache: Improving OID Indexing in Temporal Object-Oriented Database Systems

Kjetil Nørvåg
Department of Computer and Information Science
Norwegian University of Science and Technology, Norway
noervaag@idi.ntnu.no

Abstract

In a temporal OODB, an OID index (OIDX) is needed to map from OID to the physical location of the object. In a transaction time temporal OODB, the OIDX should also index the object versions. In this case, the index entries, which we call *object descriptors* (OD), also include the commit timestamp of the transaction that created the object version. The OIDX in a non-temporal OODB only needs to be updated when an object is created, but in a temporal OODB, *the OIDX has to be updated every time an object is updated*. This has previously been shown to be a potential bottleneck, and in this paper, we present the *Persistent Cache* (PCache), a novel approach which reduces the index update and lookup costs in temporal OODBs. We develop a cost model for the PCache, and use this to show that the use of a PCache can reduce the average access cost to only a fraction of the cost when not using the PCache. Even though the primary context of this paper is OID indexing in a temporal OODB, the PCache can also be applied to general secondary indexing, and can be especially beneficial for applications where updates are non-clustered.

1 Introduction

In a transaction time temporal object-oriented database system (TOODB), updating an object creates a new version of the object, but the old version is still accessible. A system maintained timestamp is associated with every object version, usually the commit time of the transaction that created this version of the object.

An important feature of OODBs, is that an object is uniquely identified by an object identifier (OID), and that the object can be accessed via its OID. The OID can be physical, which means that the disk page of the object is given directly from the OID, or logical, which means that an OID index (OIDX) is needed to map from logical OID to the physical location of the object. In a TOODB, logical OIDs is the most reasonable alternative, because an OIDX is necessary anyway to index the object versions. The entries in the OIDX, which we call *object descriptors* (OD), contain administrative information, including information to do the mapping from logical OID to physical address, and the commit timestamp. The OIDX can be quite large. In non-temporal OODBs, a typical size is in the order of 20% of the size of the database itself [2]. This means that in general, only a small part of the OIDX fits in main memory, and that OIDX retrieval can become a bottleneck if efficient access and buffering strategies are not applied.

An important difference between OIDX management in non-temporal and TOODBs, is that with only one version of an object (non-temporal), the OIDX needs only to be updated when a new object is created. This can be done in an efficient append-only operation [2], and we can focus on optimizing OIDX lookups. In a TOODB however, the OIDX must be updated *every time an object is updated*. An object update creates a new object version, without deleting the previous version, hence, a new OD for the new version has to be inserted into the OIDX. The index pages will usually have low locality (the unique part of an OID is usually an integer that will always be assigned monotonic increasing values), and as a result index updates might become a serious bottleneck in a TOODB.

To reduce disk I/O in index operations, the most recently used *index pages* are kept in an *index page buffer*. OIDX pages will in general have low locality, and to increase the probability of finding a certain OD needed for a mapping from OID to physical address, it is also possible to keep the most recently used *index entries* (the ODs) in a separate OD

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

cache, as is done in the Shore OODB [6]. With low locality on index pages, a separate OD cache utilizes memory better, space is not wasted on large pages where only small parts of them will be used. An OD cache reduces the index lookup costs considerably, and can be extended to reduce index update costs as well [10].

However, even when using a "writable" OD cache, OIDX updates are still very costly. In this paper, we present an approach to further reduce the OIDX update costs. Noting that the main reason for the bottleneck against the OIDX is the low locality of entries in the OIDX tree nodes, we use an intermediate *disk resident buffer* between the main memory buffer, and the OIDX itself. We call this the *Persistent Cache* (PCache). The PCache is typically much larger than the available main memory buffer, but smaller than the OIDX itself. The entries in the PCache are managed in an LRU like way, just like a main memory cache. In addition to reducing update costs, the PCache also reduces the lookup costs. The reason for the reduced costs, is the higher locality on PCache pages, compared to the OIDX tree nodes. Higher locality means that less disk operations are necessary to read and write ODs. The PCache-to-OIDX tree writeback can be done very efficiently later. This will be described later in this paper.

In this paper, we describe the PCache in detail, and analyze its performance by the use of cost functions. We will study optimal size of the PCache, and see how buffering of nodes in main memory should be done to optimize the PCache performance. It should also be noted that even though our primary context for this paper is OID indexing, the results are also relevant to entry access cost and index entry caching for general secondary indexes.

The organization of the rest of the paper is as follows. In Section 2 we give an overview of related work. In Section 3 we describe object and index management in TOODBs. In Section 4 we describe the PCache. In Section 5 we develop an the OID access cost model, and in Section 6 we use this cost model to study how different PCache sizes, memory sizes, index sizes, and access patterns affect the performance. Finally, in Section 7, we conclude the paper and outline issues for further research.

2 Related Work

Temporal database systems are in general still an immature technology, and in the case of transaction time TOODBs, we are only aware of one prototype[1] that has temporal OID indexing, POST/C++ [12]. The performance results presented for POST/C++ are only for relatively small databases, where the index fits in main memory, and we expect that with a larger number of objects, the OIDX would be a bottleneck.

The PCache has similarities to LHAM [7], where a hierarchy of indexes is used. One important differences between the PCache and LHAM, is that in LHAM, *all* entries in one level is regularly moved to the next, there are no LRU management, and as such, LHAM only helps in improving write efficiency, not read efficiency.

The cost models in this paper are based on previous work on modeling non-temporal OODBs [9] and temporal OODBs [10], but the models have been extended to include the aspects of the PCache. The buffer and tree models have been compared with simulation results. Detailed results from the simulations with different index sizes, buffer sizes, index page fanout, and access patterns, can be found in [11].

3 TOODB Object and Index Management

We start with a description of how OID indexing and version management can be done in a TOODB. This brief outline is not based on any existing system, but the design is close enough to make it possible to integrate into current OODBs if desired, and it will also be used as a basis for the OID indexing in the Vagabond TOODB.

3.1 Temporal OID Indexing

In a traditional OODB, the OIDX is usually realized as a hash file or a B^+-tree, with ODs as entries, and using the OID as the key. In a TOODB, we have more than one version of some of the objects, and we need to be able to access current as well as old versions efficiently. If access is mostly reading current objects, it is efficient to have two indexes, one with ODs representing the current version of the objects, and one with ODs representing historical objects (i.e., previous versions). The problem with this approach, is that every time a new version is created, we have to update *two* indexes. A second approach, is to use a linked list of versions for each object. If accesses are mostly of the type "get all versions of an object with OID i", this is an efficient alternative. However, access to a particular version, valid at time t, is very costly with this approach, because we have to traverse the object chain.

Our approach to indexing is to have *one* index structure, containing all ODs, current as well as previous versions. While several efficient multiversion access methods exist, e.g., TSB-tree [4] and LHAM [7], they are not suitable for our purpose. We will never have search for a (consecutive) range of OIDs, OID search will always be for *perfect match*, and most of them are assumed to be to the current version. TSB-trees provides more flexibility than needed, e.g., combined key range and time range search, which implies an extra cost, while LHAM can have a high lookup cost when the current version of an object is searched for.

In this paper, we assume one OD for each object version, stored in a B^+-tree. We include the commit time *TIME* in the OD, and use the concatenation of OID and time, *OID*||*TIME*, as the index key. By doing this, ODs for a particular OID will be clustered together in the leaf nodes, sorted on commit time. As a result, search for the current version of a particular OID as well as retrieval of a particular time interval for an OID can be done efficiently.

When a new object is *created*, i.e., a new OID allocated, its OD is appended to the index tree as is done in the case of

[1] Support for versioning exists in most OODBs, but not temporal management, indexing, and operations.

the Monotonic B^+-tree [3]. This operation is very efficient. However, when an object is *updated*, the OD for the new version *has to be inserted into the tree*.

It should be noted that this OIDX is inefficient for many typical temporal queries. As a result, additional secondary indexes can be needed, of which both TSB-tree and LHAM are good candidates. However, *the OIDX is still needed*, to support navigational queries, one of the main features of OODBs compared to relational database systems. Some optimizations are possible to this OIDX, e.g., using variants of nested tree index, but as the PCache is the focus of this paper, we will not elaborate more on this subject here.

3.2 Temporal Object Management

In a non-temporal (one-version) OODB, space is allocated for an object when it is created, and further updates to the object are done in-place. This implies that after an object update, the previous version of the object is not available. The physical location of the new version is the same as the previous version, hence, the OIDX needs only to be updated when objects are created and when they are deleted.

In a TOODB, it is usually assumed that most accesses will be to the current versions of the objects in the database. To keep these accesses as efficient as possible, and benefit from object clustering,[2] the database is partitioned, with current objects in one partition, and the previous versions in the other partition, in the *historical database*. When an object is updated in a TOODB, the previous version is first moved to the historical database, before the new version is stored in-place in the current database. The OIDX needs to be updated *every time an object is updated*. As long as the modified ODs are written to the log before commit, we do not need to update the OIDX itself immediately. This is done in the background, and can be postponed until the second checkpoint after the OD have been written to the log. Index pages will be written to disk either because of checkpointing, or because of buffer replacement.

Not all the data in a TOODB is temporal, for some of the objects, we are only interested in the current version. To improve efficiency, the system can be made aware of this. In this way, some of the objects can be defined as non-temporal. Old versions of these are not kept, and the objects can be updated in-place as in an one-version OODB, and the costly OIDX update is not needed when an object is modified. This is an important point: using an OODB which efficiently supports temporal data management, should not reduce the performance of applications that do not utilize these features.

4 The Persistent Index Entry Cache

The ODs accessed will be almost uniformly distributed over the index leaf nodes. The OD cache makes read accesses efficient, but in a database with many objects, most of the ODs that are updated during one checkpoint interval will reside in different leaf nodes. This low locality means that *many leaf nodes have to be updated*. When an index node is to be updated, an installation read of the node has to be done first. With a large index, the access to the nodes will be random disk accesses, and as a result, the installation read is very costly.

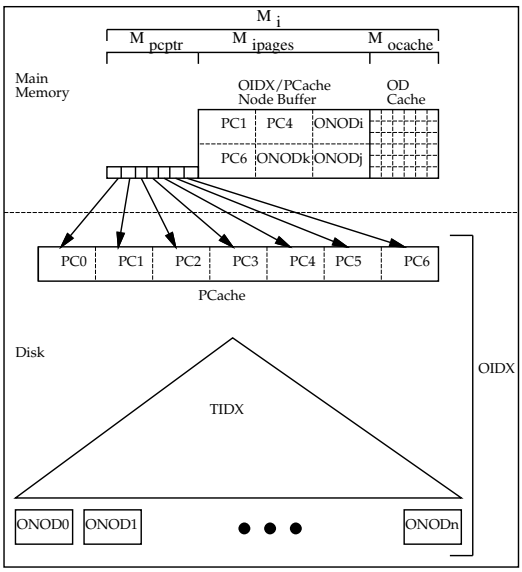


Figure 1: Overview of index, PCache, and index related main memory buffers. PCache nodes PC1, PC4 and PC6, and 3 OIDX nodes (denoted ONODn) are in the buffer.

To reduce the average access cost, the *persistent cache* (PCache) can be used. The PCache contains a subset of the entries in the OIDX, *the goal is to have the most frequently used ODs in the PCache*. In contrast to the main memory cache (the OD cache), the PCache is persistent, so that we do not have to write its entries back to the OIDX itself during each checkpoint interval. This is actually the main purpose of the PCache: to provide an intermediate storage area for persistent data, in this case, ODs.

The size of the PCache is in general larger than the size of the main memory, but smaller than the size of the OIDX. The contents of the PCache is maintained according to an LRU like mechanism. The result should be high locality on accesses to the PCache nodes, reducing the total number of installation reads, and making checkpointing less costly. Average OIDX lookup cost should also be less than without a PCache.

To avoid confusion, we will hereafter denote the index tree itself as the TIDX, and use OIDX to mean the combined index system, i.e., PCache and TIDX. Thus, when we say an entry is in the OIDX, it can be in the PCache, in

[2] It is also possible that in a TOODB application, a good object clustering includes historical objects as well as current objects. This should be studied further, but does not have any implications to the results studied here, all updates to objects will necessarily necessitate allocations of space for the new object, and an OIDX update.

the TIDX, or in both. This is illustrated on Figure 1.

4.1 PCache Organization

The index related main memory buffers, the PCache, and the TIDX, are illustrated on Figure 1. The number of nodes in the PCache should be small enough to make it possible to store pointers to all the PCache nodes in main memory. To be able to do the copying of the ODs from the PCache to the TIDX efficiently, the nodes in the PCache should be accessed in the same order as the leaf nodes in the OIDX. Therefore, the nodes in the PCache are range partitioned, each node stores a certain interval of OIDs.

Range-partitioning is vulnerable to skew. To avoid this, the partitioning can be dynamically changed (for each node, we have the OID range boundary in main memory). This is done based on the update access rates on each node. A high update access rate to a node results in a smaller interval being allocated to that node. Because the PCache nodes are frequently accessed, the repartitioning does not represent any extra cost (the repartitioning can be done when neighbor nodes are resident in the buffer).

It is important to note that even though reads of PCache nodes will be random disk accesses, the PCache nodes will be clustered together, so that the random read of PCache nodes will have a small seek time. There will also be several PCache node read requests at any time, so by using an elevator algorithm, the cost of reading from the PCache will be low.

4.2 PCache LRU Management

The PCache nodes are operated similar to an ordinary main memory cache, and when an entry is to be stored in a node in the PCache, one of the existing entries have to be discarded. To provide access statistics, an access table is maintained for each node, with one bit for each entry in the node, and we use the clock algorithm as an LRU approximation. An access bit is set each time an entry is accessed. As for the storage of the access tables, we have several options:

1. The access table of a node could be stored *in the node itself*. A problem with this option, is that when a node is to be discarded from the main memory buffer, and entries have been accessed, the node needs to be written back to disk, even if none of the entries have been changed. This is not desirable.

2. Access tables are maintained in main memory, for each main memory resident node. When a node is discarded, i.e., due to buffer replacement, the table is discarded as well. A problem with the this option, is that until enough accesses have been done to the entries, the bit map is unreliable as a way to approximate LRU, and the "wrong" entries might get discarded.

3. Access tables are maintained in main memory for *all* PCache nodes. One problem with the this approach, is that one table for each node in the PCache is needed, but because the size of each table is small (one bit for each entry in the node), this will not represent a problem as long as the PCache is not too large. If the system crashes, the contents of the access tables will be lost, and wrong caching decisions might be done when the system is restarted. This does only affect *performance* at startup time, the ODs of committed operations are always safe on disk. To reduce the amount of wrong caching decisions at startup time, we store the access table in the node as well when the node is written to disk. Note that this differs from option 1, where the node is *always* written back when it is discarded, even when there are no updates to the ODs stored in the node.

Based on the observations above, we conclude that maintaining access tables in main memory for *all* PCache nodes is the best approach.

In addition to the access tables, each node also contains a table to keep track of the status of the entries with respect to the TIDX. One *dirty* bit is needed for each entry. The dirty bit is set each time an entry is modified or inserted into the node. Only the dirty entries need to be written back to the TIDX, entries not marked as modified can be safely discarded when needed.

4.3 Update Operations

When employing the PCache, inserting ODs resulting from object *updates* are *always* done to the entry in the PCache, never directly to the TIDX. Inserting an OD into a node, implies discarding another OD from the node, based on the LRU strategy. It is preferable to discard a non-dirty entry, so that a synchronous writeback of the dirty entry is avoided. To have a high probability of non-dirty slots in the PCache node, dirty entries in the PCache are regularly copied over to the TIDX itself, asynchronously in the background. This is done efficiently by mostly sequential reading of the PCache nodes, and mostly sequential installation read and subsequent writing of the TIDX nodes.

4.4 Object Creations

Object creations are still applied directly to the TIDX. An OD resulting from an object creation is an efficient append operation into the TIDX. In the case where a new ODs is to be part of the hot set, it will usually be retrieved from the TIDX nodes before they are discarded from the buffer. These ODs will on access be inserted into the PCache.

4.5 Read Operations

Read operations belong to one of two classes: navigational (single object) read, and scan operations.

Single Object Read

Read operations are done by first checking if the entry to be accessed is in the OD cache or the PCache, if not found there, the TIDX itself is searched. The search in the PCache

might result in one disk access if the actual node is not in buffer. When using range partitioning, there is only one candidate node, so that at most one disk access will be needed. Accessing the TIDX can result in one or more disk accesses if the TIDX nodes are not in buffer.

When found, either in the PCache or the TIDX, the OD is inserted into the OD cache. If the OD was not already in the PCache, we now have several options, for example:

1. Insert the OD into the PCache immediately. Note that at this point, we are guaranteed to have the candidate PCache node resident in buffer, because we have probed it during the search for the OD. If we manage to get a high hit rate on the PCache, the optimal OD cache size might be quite small in this case. Note that the OD contains, in addition to the entries in the PCache, dirty entries resulted from update operations not yet installed into the PCache.

2. Insert the OD into the PCache only when it is to be discarded from the OD cache. In this way, we get good memory utilization, we do not have to use space for the OD both in the OD cache and in the PCache. However, in this case, we are not guaranteed to have the candidate PCache node resident in buffer, it might have been discarded since it was probed.

3. Never insert the OD into the PCache, only insert entries into the PCache when doing update operations. In this case, we rely on the OD cache to keep the most frequently accessed ODs, and use the PCache to be able to do efficient update of the TIDX. This strategy delays the update of the TIDX, and means that more entries can be collected before batch updating the TIDX.

The best option to choose, depends on access pattern. The possible installation read of option 2 can make it costly, and because ODs retrieved from read operations are not inserted into the PCache in the case of option 3, we only consider option 1 in this paper.

Scan Operations

Scan operations must be treated different from single object read operations, as one single scan operation can make the current contents of the whole PCache to be discarded. The ODs retrieved during a scan operation will in general have less chance of being used again, it is not likely that the whole collection or container to be scanned, represents a hot set. Even if this is the case, it is possible that the number of ODs retrieved during the scan, is larger than the number of ODs that fits in the PCache. In this case, if we do a new scan over the collection/container, we will have a PCache hit probability of 0. This is similar to general buffer management in the case of scan operations.

As a result, scan operations should not update the PCache, but the PCache must be consulted during read, because recently updated ODs from the actual container/collection might reside in the PCache. However, this will not be very costly, because the contents of a physical container cached in the PCache will be clustered in the PCache's pages as well, so that the extra cost of reading the relevant PCache pages is only marginal.

4.6 PCache-to-TIDX Writeback

The update of the TIDX, i.e., writing dirty entries in the PCache to the TIDX, will be done in the background. This is done by reading the PCache, and install the dirty entries of these nodes into the TIDX. This is done in segments, i.e., a number of nodes, and will be mostly sequential reading and writing. The PCache-to-TIDX writeback is a scan operation, and to avoid buffer pollution, nodes accessed during this operation should not affect the rest of the buffer contents, i.e., they should not make other nodes to be removed from the buffer.

The rate of the writeback is a tuning question. By giving it higher priority, i.e., doing more frequent writeback of PCache nodes, the probability of a PCache node being full of dirty entries is less likely. This is important, because it reduces the probability of synchronous writebacks. On the other hand, higher priority to the writeback also means that more of the disk bandwidth will be used for this purpose, because each node contains a smaller number of dirty entries.

All ODs updated since the penultimate checkpoint, and still dirty in the OD cache, needs to be installed into the PCache or TIDX during one checkpoint period. This is not the case with the PCache-to-TIDX writeback. The period between each time the contents of a particular PCache node is written back can be very long, but still short enough to avoid overflow of dirty ODs in the PCache.

4.7 Buffer Considerations

We can have a buffer shared between TIDX nodes and PCache. However, this does not necessarily give optimal performance. In some cases, it might be that TIDX accesses pollutes the buffer, resulting in a low hit rate on PCache nodes. To avoid this, we can use separate buffers, one TIDX buffer, and one PCache buffer. We can also pin a certain number of the upper TIDX levels in memory, this can be advantageous because strict use of LRU is not optimal when buffering nodes of an index tree.

5 Analytical Model

Analytical modeling in database research has mostly focused on I/O costs. This is the most significant cost factor, and in reasonable implementations, the CPU processing should go in parallel with I/O transfer, making the CPU cost "invisible". With increasing amounts of main memory available, this is not necessarily correct, but CPU costs can easily be incorporated into analytic models, and hence we consider it as an orthogonal issue to the one discussed in this paper (though it should be noted, that CPU cost should not affect the qualitative results in this paper). A more important aspect of the increasing amount of main memory,

however, is that buffer characteristics become more important, hence, the increased buffer space available must be reflected in the models.

We use a traditional disk model, where the cost of reading a block from disk is the sum of the start up cost T_{start} and the transfer cost T_{transfer}. In our model, the average start up cost is fixed, and is set equivalent to t_r, the time it takes to do one disk revolution. The transfer cost is directly proportional to the block size, and is equivalent to reading disk tracks contiguously, e.g., transfer cost is equal to $\frac{b}{V_s} t_r$, where b is the block size to be transferred, and V_s is the amount of data on one track. Thus, the total time it takes to transfer one block is $T_b(b) = T_{\text{start}} + T_{\text{transfer}} = t_r + \frac{b}{V_s} t_r$. Index costs can be reduced by partitioning the index over several disks. Declustering PCache nodes and TIDX nodes over several disks is straightforward.

The time to read or write a random index page is $T_P = T_b(S_P)$, where S_P is the index page size. In this paper, we do not consider the cost of reading and writing the objects themselves, or log operations. Those costs are independent of the indexing costs, usually done on separate disks, and are issues orthogonal to the ones studied in this paper.

In this paper, we focus on reducing access times. Obviously, the reduced access times comes at the expense of more disk space for the PCache. As disk capacity increases rapidly, with a corresponding decrease in price, we expect that in most cases, using the extra space for the PCache will be worthwhile.

As illustrated on Figure 1, a certain amount of memory, M_{ipages}, is reserved for the index page buffer, i.e., for buffering PCache and TIDX pages, and M_{ocache}, is reserved for the OD cache.

If we assume the size of each OD is S_{od}, and an overhead of S_{oh} bytes is needed for each entry in the OD cache, the number of entries that fits in the OD cache is approximately $N_{\text{ocache}} \approx \frac{M_{\text{ocache}}}{(S_{\text{od}}+S_{\text{oh}})}$. If we use an hash table and a clock algorithm as an LRU approximation, $S_{\text{oh}} \approx 8$ B (B=bytes).

The number of index pages that fits in the buffer is approximately $N_{\text{ibuf}} \approx \frac{M_{\text{ipages}}}{(S_{\text{P}}+S_{\text{oh}})}$. For each PCache page on disk, we need to keep in memory the disk address of the node (4 B), the OID range boundary (4 B), and the LRU access table as described previously. This occupies a total of $M_{\text{pcptr}} = S_{PC}(\frac{S_P/S_{\text{od}}}{8} + 8)$ B, where S_{PC} is the number of PCache nodes.

5.1 Index Entry Access Model

We assume accesses to objects in the database system to be random, but skewed (some objects are more often accessed than others). We further assume it is possible to (logically) partition the range of OIDs into partitions, where each partitions has a certain size and access probability. This is illustrated at the bottom of Figure 2 (note that this is not how it is stored on disk, this is just a model of accesses). We consider a database in a stable condition, with a total of N_{objver} objects versions (and hence, N_{objver} index entries). Note that with the TIDX described in Section 3.1,

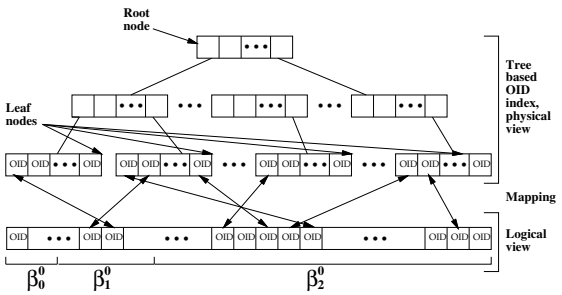


Figure 2: OID index. The lower part shows the index from a logical view, the upper part is as an index tree, which is how it is realized physically. We have indicated with arrows how the entries are distributed over the leaf nodes.

performance is not dependent of the number of existing versions of an object, only the total number of versions in the database.

In many analysis and simulations, the 80/20 model is applied, where 80% of the accesses go to 20% of the database. While this is satisfactory for analysis of some problems, it has a major shortcoming when used to estimate the number of distinct objects to be accessed. When applied, it gives a much higher number of distinct accessed objects than in a real system. The reason is that for most applications, inside the hot spot area (20% in this case), there is an even hotter and smaller area, with a much higher access probability. This is even more important for a temporal database. Most of the accesses will be to a small number of the current versions. With a large number of previous versions, this hot spot area will be much smaller and "hotter" than the one in a typical "traditional" database. This has to be reflected in the model.

5.2 Index Tree Size

If we assume an index tree with space utilization U (typically less than 0.69 for a B$^+$-tree), the number of leaf nodes is $N^0_{\text{tree}} \approx \frac{N_{\text{objver}}}{U\lfloor S_P/S_{\text{od}}\rfloor}$. The fanout F of internal nodes is $F = \lfloor US_P/S_{\text{ie}}\rfloor$, where S_{ie} is the size of an entry in an internal node. The number of levels in the tree is $H = 1+\lceil \log_F N^0_{\text{tree}}\rceil$. The number of nodes at each level in the tree is $N^i_{tree} = \lceil \frac{N^{i-1}_{\text{tree}}}{F}\rceil$ and the total number of nodes in the tree is $N_{\text{tree}} = \sum_{i=0}^{H-1} N^i_{\text{tree}}$.

5.3 Buffer Performance Model

Our buffer model is an extension of the Bhide, Dan and Dias LRU buffer model (BDD) [1]. Due to space constraints, we only present the most important aspects of our model in this paper, but a detailed description can be found in [9].

A database in the BDD model has a size of N data granules (e.g., pages), partitioned into p partitions. Each par-

tition contains a fraction β_i of the data granules, and α_i of the accesses are done to each partition. The distributions *within* each of the partitions are assumed to be uniform, and all accesses are assumed to be independent. We denote a particular partitioning set $\Pi = (\alpha_0, \ldots, \alpha_{p-1}, \beta_0, \ldots, \beta_{p-1})$. For example, for the 80/20 model, $\Pi_{80/20} = (0.8, 0.2, 0.2, 0.8)$. We will in the following use Π_A as short for the actual OD access partitioning set.

In the BDD model, the steady state average buffer hit probability is denoted $P_{\text{buf}}(B, N, \Pi)$, where B is the number of data granules that fits in the buffer. The buffer hit probability for data granules belonging to a particular partition p is denoted $P^p_{\text{buf}}(B, N, \Pi)$. The BDD model can also be used to calculate the total number of distinct data granules accessed after n accesses to the database, $N_{\text{distinct}}(n, N, \Pi)$.

OD Cache Hit Rate

Accesses to the OD cache can be assumed to follow the assumptions behind the BDD model, they are independent and random requests. By applying this model with object entries as data granules, the probability of an OD cache hit is $P_{\text{ocache}} = P_{\text{buf}}(N_{\text{ocache}}, N_{\text{objver}}, \Pi_A)$.

General Index Buffer Model

The BDD LRU buffer model only models independent, non-hierarchical, access. Modeling buffer for hierarchical access is more complicated. Even though searches to the leaf pages can be considered to be random and independent, nodes accessed during traversal of the tree are *not* independent. We have extended the original model to be able to analyze buffer performance in the case of hierarchical index accesses as well [11]. This is based on the observation that *each level* in the tree is accessed with the *same probability* (assuming traversal from root to leaf on every search). Thus, with a tree with H levels, we initially have H partitions. Each of these partitions are of size N^i_{tree}, where N^i_{tree} is the number of index pages on level i in the tree. The access probability is $\frac{1}{H}$ for each partition.

To account for hot spots, we further divide the leaf page partition into p' partitions, each with a fraction of β_{Li} of the leaf nodes, and access probability α_{Li} relative to the other leaf page partitions. Thus, in a "global" view, each of these partitions have size $\beta_{Li} N^0_{\text{tree}}$ and access probability $\frac{\alpha_{Li}}{H}$. In total, we have $p = p' + (H - 1)$ partitions. The hot spots at the leaf page level make access to nodes on upper levels non-uniform, but as long as the fanout is sufficiently large, and the hot spot areas are not too narrow, we can treat accesses to nodes on upper levels as uniformly distributed within each level. An example of this partitioning is illustrated to the right on Figure 3, where a tree with $H = 4$ levels and $p' = 2$ leaf page partitions is partitioned into $p = 2 + (4 - 1) = 5$ partitions.

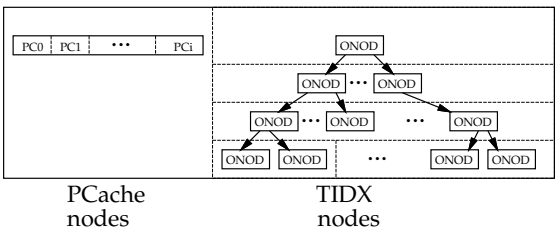


Figure 3: Access partitions.

Index Page Access Model

As noted, we can assume low locality in index pages. Because of the way OIDs are generated, entries from a certain partition are not clustered in the index. This is illustrated in Figure 2, where a leaf node containing index entries contains unrelated entries from different partitions. This means that the access pattern for the leaf nodes is different from the access pattern to the database from a logical view. As described in [11], we can use the initial OD partitioning (the OD access pattern) Π_A as basis for deriving the leaf node access partitioning Π_L.

PCache Hit Rate

We denote the probability that a certain OD is in the PCache as P_{PC}, the actual PCache page might be in memory or on disk.

The number of ODs in each PCache node is $\lfloor \frac{S_P}{S_{\text{od}}} \rfloor$, and with a total of S_{PC} nodes, the number of ODs that fits in the PCache is $N_{PC} = S_{PC} \lfloor \frac{S_P}{S_{\text{od}}} \rfloor$. The probability that a certain OD is in one of the PCache nodes can be approximated to:

$$P_{PC} = P_{\text{buf}}(N_{PC}, N_{\text{objver}}, \Pi_A)$$

PCache and TIDX Buffer Model

When the PCache and the TIDX share the same main memory page buffer, the model has to reflect this. The access probabilities for TIDX nodes and PCache nodes are different, and as a result, in an LRU managed buffer, the hit rate will be different. In the buffer model, we use a partitioning as illustrated on Figure 3. On the figure, the PCache is one partition, and each level in the tree is one partition, with the leaf node partition further divided into two partitions, reflecting the existence of hot spot nodes (nodes belonging to one of the two leaf node partitions need not actually be physically adjacent as on the figure).

Considering the page accesses, all OIDX lookups will access one PCache page, and $(1 - P_{PC})$ of the lookups will also access the TIDX. Each OIDX lookup results on average in $1 + (1 - P_{PC})H$ page accesses. Thus, the total access probability of the PCache is $\alpha_{PC} = \frac{1}{1+(1-P_{PC})H}$, which is the fraction of accessed pages that is part of the PCache partition. The TIDX access probability is $\alpha_{TIDX} = \frac{(1-P_{PC})H}{1+(1-P_{PC})H}$.

The total number of index pages is $S_{PC} + N_{tree}$. The PCache contains $\beta_{PC} = \frac{S_{PC}}{S_{PC}+N_{tree}}$ of these pages, the TIDX contains $\beta_{TIDX} = \frac{N_{tree}}{S_{PC}+N_{tree}}$ of the pages.

The TIDX partitions is further partitioned into p partitions as described above, and we denote the resulting partitioning (Figure 3) as Π_{Shared}. We denote the PCache and TIDX node buffer hit probabilities as:

$$P_{\text{buf_PC}} = P_{\text{buf}}^{PC}(N_{\text{ibuf}}, S_{PC} + N_{\text{tree}}, \Pi_{\text{shared}})$$

$$P_{\text{buf_TIDX}} = P_{\text{buf}}^{TIDX}(N_{\text{ibuf}}, S_{PC} + N_{\text{tree}}, \Pi_{\text{shared}})$$

As noted in Section 4.7, it can be advantageous to use separate buffers for the PCache and TIDX. In that case, $N_{\text{ibuf_PC}}$ buffer pages are reserved for the PCache, and $N_{\text{ibuf_TIDX}}$buffer pages are reserved for the TIDX, so that $N_{\text{ibuf_PC}} + N_{\text{ibuf_TIDX}} = N_{\text{ibuf}}$. Denoting the tree partitioning as Π_{Tree}, the corresponding buffer hit probabilities using separate buffers are:

$$P_{\text{buf_PC}} = \frac{N_{\text{ibuf_PC}}}{S_{PC}}$$

$$P_{\text{buf_TIDX}} = P_{\text{buf}}(N_{\text{ibuf_TIDX}}, N_{tree}, \Pi_{\text{Tree}})$$

5.3.1 OIDX Lookup Cost

Assuming we have the address of all PCache pages in memory, and use a range partitioned PCache, at most one disk access is needed for each PCache lookup. Before a page can be read in, another have to be replaced. A page may contain dirty entries, because all read ODs from the TIDX are inserted immediately. In this case, the PCache page has to be written back. To be able to do this efficiently, we use the following strategy: On disk, we allocate space for more nodes than the number of nodes in the PCache. When we read a page, we at the same time schedule dirty page(s) for writing, to an empty slot near the node(s) to be read. In that way, the extra write cost is only marginal compared to the read. Because we at all times keep the pointers to all PCache nodes in the memory, they can be written back to different places every time.[3] We approximate the lookup cost to:

$$T_{\text{lookup_PC}} = (1 - P_{\text{buf_PC}})T_P$$

To access an entry in the TIDX, it is necessary to traverse the H levels from the root to a leaf page. To do this, we need $(1 - P_{\text{buf_TIDX}})H$ disk accesses. The average cost of an TIDX lookup is:

$$T_{\text{lookup_TIDX}} = (1 - P_{\text{buf_TIDX}})HT_P$$

With a probability of P_{ocache}, the OID entry requested is already in the OD cache, but for $(1 - P_{\text{ocache}})$ of the requests, we have to access the PCache. With a probability of P_{PC}, the entry is in the PCache. If not, the TIDX itself has to be accessed. The average cost to retrieve an entry is:

$$T_{\text{lookup}} = (1 - P_{\text{ocache}})(T_{\text{lookup_PC}} + (1 - P_{PC})T_{\text{lookup_TIDX}})$$

[3] It is interesting to note that by using this approach, we do things according to the log-structured file system philosophy, which our Vagabond TOODB will be based on [8].

5.3.2 OIDX Update Cost

We do not need to update the PCache or the index pages in the TIDX immediately after an update has been done. This is done in the background, and can be postponed, increasing the probability that several updates can be done to each index page before they are written back. We calculate the average index update cost as the total index update costs during an interval, divided on the number of updates. In this context, we define the checkpoint interval to be the number of objects that can be written between two checkpoints. The number of written objects, N_{CP}, includes created as well as updated objects. $P_{\text{new}}N_{CP}$ of the written objects are creations of new objects, and $(1 - P_{\text{new}})N_{CP}$ of the written objects are updates of existing objects. We assume that the OD cache is large enough to keep all dirty ODs through one checkpoint interval, and that deleting and compacting pages can be done in background. This means that $N_{CP} < N_{\text{ocache}}$. Using a strategy that write a larger amount of ODs to the log before installing them into the TIDX, is difficult: If we did not keep all ODs not yet installed into the OIDX in the memory, we would have to search the log on each access, to check if the log contained a more recent version than the one in the OIDX.

Creation of New ODs

New object descriptors are created when new objects are created. The number of created objects is $N_{CR} = P_{\text{new}}N_{CP}$. When new objects are created, their ODs are appended to the index (we do not distribute the entries over the old node and the new one when the rightmost nodes are split), and we have clustered updates. As described previously, object creations are done directly to the TIDX, and not to the PCache. This contributes to $N_n^0 = \frac{N_{CR}}{\lfloor S_P/S_{\text{od}} \rfloor}$ created leaf pages. This is a subtree in the index tree, of height H_s, with $S_n = \sum_{i=0}^{H_s-1} N_n^i$ pages. The total cost of creating these object descriptors is the cost of writing S_n index pages to the disk. No installation read is needed for these pages. Assuming that the disk is not too fragmented, these pages can be written in one operation, mostly sequentially:

$$T_{\text{writenew}} = T_b(S_n S_P)$$

Modification of Existing ODs in the PCache

When an object is updated, a new object version is created, and a new OD has to be inserted into the OIDX. The number of updated objects during one checkpoint interval is $N_U = N_{CP} - N_{CR}$.

In general, at least one OD will be inserted into each PCache page (the number of updates in one checkpoint interval is much larger than the number of PCache pages). In this case, the most efficient way to update the PCache is to read sequentially a number of PCache pages, update them,

write them back, and continue with the next segment. Assuming that PCache-to-TIDX writeback has high enough priority, we can assume that when inserting a new entry into a PCache node, there is always a non-dirty entry that can be removed. The cost of this is:

$$T_{\text{write_to_PC}} = 2T_b(S_{PC}S_P)$$

where S_{PC} is the number of PCache nodes.

PCache-to-TIDX Writeback Cost

The purpose of the PCache-to-TIDX writeback is to always have non-dirty slots in the PCache nodes, where new entries can be inserted. The PCache-to-TIDX writeback runs continuously in the background. The period for each round is ideally so long that each PCache node is almost full of dirty entries when it is processed.

The cost is equal to reading a number of PCache nodes (sequential reading), and writing the dirty entries back to the TIDX. If we assume each PCache node is almost full when we process it, we have fN_{PC} entries to write back in each round, where f is the PCache node dirty fill factor, i.e., the amount of dirty entries in the node. This value should ideally be close to 1, but to avoid delays in normal processing due to overflow of dirty entries in nodes, f should be sufficiently small, we will in the calculations in this paper use a value of 0.90. The number of update objects during each round of PCache-to-TIDX writeback is $N_{SCP} = fN_{PC}$, which we call the *super checkpoint period.*

Updating the index involves a page installation read, where the page where the last (current) version resides is read from disk, if the page is not already in the buffer. The cost of this is $T_{\text{lookup_TIDX}}$ for each *distinct* object modified. The number of distinct updated objects is:

$$N_{DU} = N_{\text{distinct}}(N_{SCP}, N_{\text{objver}}, \Pi_A)$$

However, as noted in Section 3.2, not all objects in a TOODB are temporal. We denote the fraction of the data accesses going to temporal objects as P_{temporal}. Only updates of these objects alter the OIDX, updates of non-temporal objects will be done in-place. The number of distinct updated temporal objects is:

$$N^V_{DU} = P_{\text{temporal}}N_{DU}$$

The number of leaf pages to be accessed as a part of the installation read:

$$N_m = N_{\text{distinct}}(N^V_{DU}, N^0_{tree}, \Pi_L)$$

If there is space for the new OD in the leaf node of the TIDX, it can be inserted there, and the node can be written back. If there is no space in the node, the node is split, a process done recursively, possibly to the root. If a node is split, the parent node has to be updated as well. Because of the possibility of page splits, determining the update cost is difficult. With sufficiently many entries on each index node, the probability of page split is small enough to be neglected [13]. However, for some pages, there are more than one insertion to that page (possibly generated by several updates to one object during one checkpoint interval, remember that *each update creates a new entry to be inserted into the OIDX*). Thus, we include the page split in our cost functions. According to Loomis [5], the probability of a split in a B$^+$-tree of order m is less than $\frac{1}{\lceil m/2 \rceil - 1}$, so we approximate $P_{\text{split}} \approx \frac{1}{\lceil (S_P/S_{\text{od}})/2 \rceil - 1}$. For each split, the new page needs to be written back, as well as the updated parent node. However, note that there may be several splits affecting one parent node, in this case, it needs only be written back once. The resulting total write back cost is:

$$\begin{aligned} T'_{\text{writeback}} &= (N_m + N_m 2P_{\text{split}})T_P \\ &= N_m(1 + 2P_{\text{split}})T_P \end{aligned}$$

The equation above assumes that there will on average be less than one entry to be inserted in each leaf node. That is the case as long as we use the following optimization: If the checkpoint interval is sufficiently large, it is more efficient to read the complete index, update the index nodes, and write it back (if memory is not large enough, this is done in segments). This will be very efficient, as the reading and writing will be sequential. The cost of this is:

$$T''_{\text{writeback}} = 2T_b(N_{\text{tree}}S_P)$$

giving:

$$T_{\text{writeback}} = \min(N_m T_{\text{lookup_}TIDX} + T'_{\text{writeback}}, T''_{\text{writeback}})$$

Average Index Update Cost

The average index update cost per object is the total cost of updating the PCache and the PCache-to-TIDX writeback, divided on the number of updated objects:

$$T_{\text{update}} = \frac{T_{\text{writenew}}}{N_{CP}} + \frac{T_{\text{write_to_PC}}}{N_{CP}} + \frac{T_{\text{writeback}}}{N_{SCP}}$$

Note that the total PCache-to-TIDX writeback is for one super checkpoint period, while the PCache update and object creating cost is per ordinary checkpoint period.

5.4 OIDX Access Cost Without PCache

In a system with no PCache, the OIDX lookup cost is:

$$T_{\text{lookup}} = (1 - P_{\text{ocache}})T_{\text{lookup_TIDX}}$$

Without a PCache, we need to write back all ODs to the TIDX each checkpoint interval. This is similar to PCache-to-TIDX writeback, except that we now write back only $N_{CP} - N_{CR}$ entries instead of N_{SCP}, which makes it more difficult to do it efficiently. The average update cost is:

$$T_{\text{update}} = \frac{T_{\text{writenew}}}{N_{CP}} + \frac{T_{\text{writeback_TIDX}}}{N_{CP}}$$

Set	β_0^0	β_1^0	β_2^0	α_0^0	α_1^0	α_2^0
3P1	0.01	0.19	0.80	0.64	0.16	0.20
3P2	0.001	0.049	0.95	0.80	0.19	0.01
2P8020	0.20	0.80	-	0.80	0.20	-
2P9505	0.05	0.95	-	0.95	0.05	-

Table 1: Partition sizes and partition access probabilities for the partitioning sets used in this study.

where $T_{\text{writeback_TIDX}}$ is calculated according to the equations used for $T_{\text{writeback}}$, except that $N_{CP} - N_{CR}$ is used instead of N_{SCP} in calculating $N_D U$. When calculating the buffer hit probabilities, the index page buffer is only used for the TIDX, with the absence of a PCache, no memory is used for PCache pointers and tables.

6 Performance Study

We have now derived the cost functions necessary to calculate the average cost of OIDX access under different system parameters and access patterns, with and without the use of a PCache. We will in this section study how different values for these parameters affects the access cost, under which conditions using a PCache is beneficial, and optimal sizes for the PCache. The mix of updates and lookups to the OIDX affects the optimal parameter values, and they should be studied together. If we denote the probability that an operation is a write as P_{write}, the average index access cost is the average of the cost of all index lookup and index update operations:

$$T_{\text{access}} = (1 - P_{\text{write}})T_{\text{lookup}} + P_{\text{write}}T_{\text{update}}$$

Our goal here is to minimize T_{access}. We measure the gain from using a PCache, with optimal parameter values, as:

$$\text{Gain} = 100\left(\frac{T_{\text{access_noPCache}} - T_{\text{access}}}{T_{\text{access}}}\right)$$

where $T_{\text{access_noPCache}}$ is the access cost when not using a PCache. In the rest of this paper, we give PCache size as a fraction of the TIDX size.

It is difficult to know what kind of access pattern that will be experienced in TOODBs. It is possible to do predictions based on current access patterns, but we believe that it is quite possible that when support for temporal features become common, application developers can utilize these in new ways. The access patterns used in this paper do not necessarily represent any of these, but we will use them to show that the gain from using the PCache is considerable, under most conditions and access patterns.

We have used four access patterns. The partition sizes and access probabilities are summarized in Table 1 (note that this is the *OID* access pattern Π_A, and not the *index page* access pattern Π_L). In the first partitioning set, we have three partitions, extensions of the 80/20 model, but with the 20% hot spot partition further divided, into a 1% hot spot area, a 19% less hot area, and a 80% relatively cold area. The second partitioning set resembles the access pattern close to what we expect it to be in future TOODBs, with a large cold set, consisting of old versions. The two other sets in this analysis have each two partitions, with hot spot areas of 20% and 5%.

Parameter	Value	Parameter	Value
M_{ocache}	0.1 M	N_{objver}	100 mill.
V_s	50 KB	U	0.67
t_r	8.33 ms	N_{CP}	$0.9N_{\text{ocache}}$
S_P	8 KB	P_{new}	0.2
S_{od}	32 B	P_{write}	0.2
S_{oh}	8 B	P_{temporal}	0.8
S_{ie}	16 B		

Table 2: Default parameters.

Unless otherwise noted, results and numbers in the next sections are based on calculations using default parameters as summarized in Table 2.[4] Note that in this paper, when we talk about available main memory, we only consider the memory available for index related buffering, M_i. Main memory for object page buffering is orthogonal to this issue.

With the values in Table 2, the ODs would occupy ≈ 3.1 GB if stored compactly. Typically, the objects themselves occupies at least four times as much space as the OIDX, if this is reflected in available main memory buffer, $M_i = 50$ MB should imply a total buffer memory of 200-300 MB. In this study, we mainly investigate the index memory interval from $M_i = 1$ MB to $M_i = 50$ MB, as this is the most dynamic area of OIDX access cost, but we will also show how the availability of larger amounts of memory affects performance.

6.1 The Effect of Using a PCache

Figure 4 illustrates the typical cost involved in OIDX access, using the default parameters, but with different index memory sizes M_i. The gain is from 20% to several 100%. We see that the PCache is especially beneficial with relative small index memory sizes compared to the total index size. As the index size increases, the gain decreases (but as we will show in Section 6.4, the gain actually increases again with larger main memory sizes).

6.2 Optimal PCache Size

The optimal PCache sizes are illustrated to the left on Figure 5. With access pattern 2P8020 and 3P1, most of the

[4] Note that even though some of the parameter combinations in the following sections are unlikely to represent the average over time, they can occur in periods, e.g., more write than read operations. It is in situations like this that adaptive self tuning systems would be interesting, when parameter sets differs from the average, which systems traditionally have been tuned against.

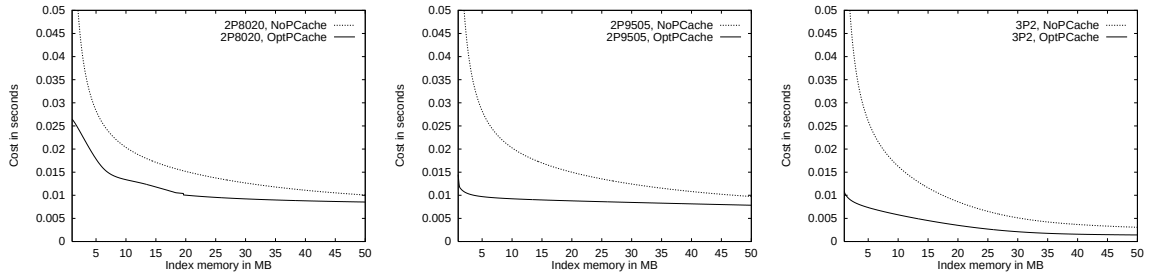


Figure 4: OIDX access cost with and without employing a PCache, with access patterns according to 2P8020, 2P9505, and 3P3.

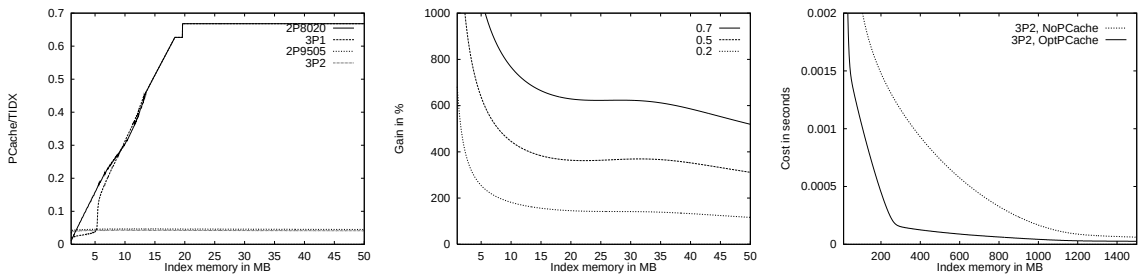


Figure 5: Optimal PCache size for different access patterns to the left. In the middle, the effect of different update ratios P_{write} for access pattern 3P2. To the right, we have the OIDX access cost with and without employing a PCache, employing large index memory sizes.

available index memory, except the memory reserved for the OD cache, is used to store the pointers and LRU tables for the PCache. 2P9505 and 3P2 have more emphasized hot spot ares, in this case, a smaller PCache is optimal, just enough to store the hot spot area.

6.3 The Effect of Different Update Ratios

The main purpose of the PCache is to increase OIDX update performance. This is illustrated very well on the middle subfigure of Figure 5, which shows the gain using different values for P_{write}.

6.4 The Effect of Larger Amounts of Memory

The right subfigure of Figure 5 illustrates that using a PCache is also beneficial when a large main memory buffer is available. The minimum gain here is when $M_i \approx 80$ MB. At that point, the gain is 94%. It then increases again, until $M_i \approx 300$ MB, where the gain is over 600%. After that, the gain from using PCache slowly decreases, with increasing amounts of available main memory.

6.5 The Effect of Different Page Sizes

The page size is an important factor in determining the indexing performance. The optimal page size is a compromise of two contradicting factors. Because of low locality, large page sizes in an OIDX means more wasted space in the index page buffer, and the optimal page size is thus much smaller. However, small pages also results in a higher tree. Even though in most cases upper levels of the index tree will be resident in memory, a tree with smaller page size also needs more space, reducing the buffer hit probability. We can see that there are two strategies for efficiency: Either large pager, which is particularly advantageous for the creation of objects, and small pages, to capture the fact that there is low level of sharing. We have studied optimal page sizes for the different access patterns, with possible page sizes between 2 KB and 64 KB. All shows that a small page size, less than the 8 KB blocks commonly used, is beneficial.

6.6 PCache Using Separate Buffers

We have also done the analysis with separate buffers for the PCache and TIDX. The analysis shows that a cost reduction of a few percent, typically from 2 to 3%, can be found. However, in this case, it is very important with accurate buffer partitioning. This assumes knowledge of current access pattern at all times, something which is difficult in practice. LRU buffer management is in this sense self

adaptive, and with only marginal improvement when using separate buffer, we advice against using separate buffers.

7 Conclusions and Future Work

We have in this paper described the PCache, and how it can be used to improve performance in an TOODB by reducing the number of disk operations needed for index maintenance. We have developed cost models which we have used to analyze the improved performance and characteristics of the PCache, under different access patterns, and memory and index sizes. The results show that:

1. The OID indexing cost in a TOODB will be large, but can be reduced by the use of a PCache.
2. The gain from using a PCache can be large.
3. The gain is especially good when using an *optimal* size of the PCache. Having an optimally tuned system is important. Access pattern in a database system is dynamic, and the system should be able to detect this, and tune the size of index page buffer and OD cache size accordingly. The cost models in this paper can be of valuable use for optimizers and automatic tuning tools in TOODBs.

In this paper, we have described several strategies for PCache LRU table storage, and PCache update strategies when doing read operations. These issues are interesting further work. It is possible that by combining several strategies in a dynamic adaptive PCache, performance can be improved even more, and making the system less vulnerable to rapidly changing access patterns and variants of data skew.

This paper described the PCache used to improve OID index in TOODBs. The PCache should also be applicable to general secondary indexing, especially interesting is applications where updates are not clustered, i.e., have low locality.

Acknowledgments

I would like to thank Olav Sandstå and Kjell Bratbergsengen for useful discussions and constructive comments.

References

[1] A. K. Bhide, A. Dan, and D. M. Dias. A simple analysis of the LRU buffer policy and its relationship to buffer warm-up transient. In *Proceedings of the Ninth International Conference on Data Engineering*, 1993.

[2] A. Eickler, C. A. Gerlhof, and D. Kossmann. Performance evaluation of OID mapping techniques. In *Proceedings of the 21st VLDB Conference*, 1995.

[3] R. Elmasri, G. T. J. Wuu, and V. Kouramajian. The time index and the monotonic B^+-tree. In A. U. Tansel, J. Clifford, S. K. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal databases: theory, design and implementation*. The Benjamin/Cummings Publishing Company, Inc., 1993.

[4] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD*, 1989.

[5] M. E. Loomis. *Data Management and File Structures*. Prentice Hall, 1989.

[6] M. L. McAuliffe. *Storage Management Methods for Object Database Systems*. PhD thesis, University of Wisconsin-Madison, 1997.

[7] P. Muth, P. O'Neil, A. Pick, and G. Weikum. Design, implementation, and performance of the LHAM log-structured history data access method. In *Proceedings of the 24th VLDB Conference*, 1998.

[8] K. Nørvåg and K. Bratbergsengen. Log-only temporal object storage. In *Proceedings of the 8th International Workshop on Database and Expert Systems Applications, DEXA'97*, 1997.

[9] K. Nørvåg and K. Bratbergsengen. An analytical study of object identifier indexing. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications, DEXA'98*, 1998.

[10] K. Nørvåg and K. Bratbergsengen. Optimizing OID indexing cost in temporal object-oriented database systems. In *Proceedings of the 5th International Conference on Foundations of Data Organization, FODO'98*, 1998.

[11] K. Nørvåg and K. Bratbergsengen. An analytical study of object identifier indexing. Technical Report IDI 4/98, Norwegian University of Science and Technology, 1998. Available from http://www.idi.ntnu.no/grupper/DB-grp/.

[12] T. Suzuki and H. Kitagawa. Development and performance analysis of a temporal persistent object store POST/C++. In *Proceedings of the 7th Australasian Database Conference*, 1996.

[13] J. D. Ullman. *Principles of database and knowledge-base systems*. Computer Science Press, 1988.

Appendix F

Efficient Use of Signatures in Object-Oriented Database Systems

This appendix contains the paper presented at Advances in Databases and Information Systems (ADBIS'99), held in Maribor, Slovenia, September 1999.

Efficient Use of Signatures in Object-Oriented Database Systems

Kjetil Nørvåg

Department of Computer and Information Science
Norwegian University of Science and Technology
7491 Trondheim, Norway
email: noervaag@idi.ntnu.no

Abstract. Signatures are bit strings, generated by applying some hash function on some or all of the attributes of an object. The signatures of the objects can be stored separately from the objects themselves, and can later be used to filter out candidate objects during a perfect match query. In an object-oriented database system (OODB) using logical object identifiers (OIDs), an object identifier index (OIDX) is needed to map from logical OID to the physical location of the object. In this paper we show how signatures can be stored in the OIDX, and used to reduce the average object access cost in a system. We also extend this approach to transaction time temporal OODBs (TOODB), where this approach is even more beneficial, because maintaining signatures comes virtually for free. We develop a cost model that we use to analyze the performance of the proposed approaches, and this analysis shows that substantial gain can be achieved.

1 Introduction

A signature is a bit string, which is generated by applying some hash function on some or all of the attributes of an object.[1] When searching for objects that match a particular value, it is possible to decide from the signature of an object whether the object is a possible match. The size of the signatures is generally much smaller than the size of the objects themselves, and they are normally stored separately from the objects themselves, in signature files. By first checking the signatures when doing a perfect match query, the number of objects to actually be retrieved can be reduced.

Signature files have previously been shown to be an alternative to indexing, especially in the context of text retrieval [1, 6]. Signature files can also be used in general query processing, although this is still an immature research area.

The main drawback of signature files, is that signature file maintenance can be relatively costly. If one of the attributes contributing to the signature in an object is modified, the signature file has to be updated as well. To be beneficial, a high read to write ratio is necessary. In addition, high selectivity is needed at query time to make it beneficial to read the signature file in addition to the objects themselves.

[1] Note that *signatures* are also often used in other contexts, e.g., function signatures and implementation signatures.

In this paper, we first show how signatures can be incorporated into traditional object-oriented databases (OODB). Second, we show how they can be used in *transaction time temporal OODBs*, with only marginal maintenance cost.

Every object in an OODB is uniquely identified by an object identifier (OID). To do the mapping from logical OID to physical location, an OID index (OIDX), often a B-tree variant, is used.[2] The entries in the OIDX, which we call *object descriptors* (OD), contains the physical address of the object. Because of the way OIDs are generated, OIDX accesses often have low locality, i.e., often only one OD in a particular OIDX leaf node is accessed at a time. This means that OIDX lookups can be costly, but they have to be done every time an object is to be accessed (as will be explained later, the lookup cost can be reduced by employing OD caching). OIDX updates are only needed when object are created, moved, or deleted. It is not necessary when objects are updated, because updates to object are done in-place, so that the mapping information in the OIDX is still valid after an object have been updated.

Our approach to reduce the average access cost in the system, is to include the signature of an object *in the OIDX itself*. This means that the OD now also includes the signature, in addition to the mapping information. When we later do a value based perfect match search on a set, we can in many cases avoid retrieving the objects themselves, checking the signature in the OD is enough to exclude an object during the search. The OD will have to be retrieved anyway, because it is needed to find the physical location of the object, so there is no additional cost to retrieve the OD, compared to not using signatures. Storing the signature in the OIDX increases the size of the OD, and the size of the OIDX, and makes an OIDX update necessary every time an object is updated, but as we will show later in this paper, in spite of this extra cost, it will in most cases be beneficial.

A context where storing signatures in the OIDX is even more interesting, is transaction time temporal OODBs (TOODB). In a TOODB, object updates do not make previous versions unaccessible. On the contrary, previous versions of objects can still be accessed and queried. A system maintained timestamp is associated with every object version. This timestamp is the commit time of the transaction that created this version of the object. In a non-temporal OODB, the OIDX update would not be necessary if we did not want to maintain signatures. In a TOODB, on the other hand, *the OIDX must be updated every time an object is updated*, because we add a new version, and the timestamp and the physical address of the new version need to be inserted into the index. As a result, introducing signatures only marginally increases the OIDX update costs. Because of the low locality on updates, disk seek time dominates, and the increased size of the ODs is of less importance. *With this approach, we can maintain signatures at a very low cost, and by using signatures, one of the potential bottlenecks in an TOODB, the frequent and costly OIDX updates, can be turned into an advantage!*

The organization of the rest of the paper is as follows. In Sect. 2 we give an overview of related work. In Sect. 3 we give a brief introduction to signatures. In Sect. 4 we describe indexing and object management in a TOODB. In Sect. 5 we describe how

[2] Some OODBs avoid the OIDX by using physical OIDs. In that case, the OID gives the physical disk page directly. While this potentially gives a higher performance, it is very inflexible, and makes tasks as reclustering and schema management more difficult.

signatures can be integrated into OID indexing. In Sect. 6 we develop a cost model, which we use in Sect. 7 to study the performance when using signatures stored in the OIDX, with different parameters and access patterns. Finally, in Sect. 8, we conclude the paper and outline issues for further research.

2 Related Work

Several studies have been done in using signatures as a text access methods, e.g. [1, 6]. Less has been done in using signatures in ordinary query processing, but studies have been done on using signatures in queries on set-valued objects [7].

We do not know of any OODB where signatures have been integrated, but we plan to integrate the approaches described in this paper in the Vagabond parallel TOODB [9].

3 Signatures

In this section we describe signatures, how they can be used to improve query performance, how they are generated, and signature storage alternatives.

A signature is generated by applying some hash function on the object, or some of the attributes of the object. By applying this hash function, we get a signature of F bits, with m bits set to 1. If we denote the attributes of an object i as $A_1, A_2, \ldots, A_n$, the signature of the object is $s_i = S_h(A_j, \ldots A_k)$, where S_h is a hash value generating function, and $A_j, \ldots A_k$ are some or all of the attributes of the object (not necessarily including all of $A_j, \ldots A_k$). The size of the signature is usually much smaller than the object itself.

3.1 Using Signatures

A typical example of the use of signatures, is a query to find all objects in a set where the attributes match a certain number of values, $A_j = v_j, \ldots, A_k = v_k$. This can be done by calculating the query signature s_q of the query: $s_q = S_h(A_j = v_j, \ldots, A_k = v_k)$ The query signature s_q is then compared to all the signatures s_i in the signature file to find possible matching objects. A possible matching object, a *drop*, is an object that satisfies the condition that all bit positions set to 1 in the query signature, also are set to 1 in the object's signature. The drops forms a set of candidate objects. An object can have a matching signature even if it does not match the values searched for, so all candidate objects have to be retrieved and matched against the value set that is searched for. The candidate objects that do not match are called *false drops*.

3.2 Signature Generation

The methods used for generating the signature depend on the intended use of the signature. We will now discuss some relevant methods.

Whole Object Signature. In this case, we generate a hash value from the whole object. This value can later be used in a perfect match search that includes all attributes of the object.

One/Multi Attribute Signatures. The first method, *whole object signature*, is only useful for a limited set of queries. A more useful method is to create the hash value of only one attribute. This can be used for perfect match search on that specific attribute. Often, a search is on perfect match of a subset of the attributes. If such searches are expected to be frequent, it is possible to generate the signature from these attributes, again just looking at the subset of attributes as a sequence of bits. This method can be used as a filtering technique in more complex queries, where the results from this filtering can be applied to the rest of the query predicate.

Superimposed Coding Methods. The previous methods are not very flexible, they can only be used for queries on the set of attributes used to generate the signature. To be able to support several query types, that do perfect match on different sets of attributes, a technique called *superimposed coding* can be used. In this case, a separate attribute signature for each attribute is created. The object signature is created by performing a bitwise OR on each attribute signature, for an object with 3 attributes the signature is $s_i = S_h(A_0)$ OR $S_h(A_1)$ OR $S_h(A_2)$. This results in a signature that can be very flexible in its use, and support several types of queries, with different attributes.

Superimposed coding is also used for fast text access, one of the most common applications of signatures. In this case, the signature is used to avoid full text scanning of each document, for example in a search for certain words occurring in a particular document. There can be more than one signature for each document. The document is first divided into logical blocks, which are pieces of text that contain a constant number of distinct words. A word signature is created for each word in the block, and the block signature is created by OR-ing the word signatures.

3.3 Signature Storage

Traditionally, the signatures have been stored in one or more separate files, outside the indexes and objects themselves. The files contains s_i for all objects i in the relevant set. The sizes of these files are in general much smaller than the size of the relation/set of objects that the signatures are generated from, and a scan of the signature files is much less costly than a scan of the whole relation/set of objects. Two well-know storage structures for signatures are *Sequential Signature Files* (SSF) and *Bit-Sliced Signature Files* (BSSF). In the simplest signature file organization, SSF, the signatures are stored sequentially in a file. A separate *pointer file* is used to provide the mapping between signatures and objects. In an OODB, this file will typically be a file with OIDs, one for each signature. During each search for perfect match, the whole signature file has to be read. Updates can be done by updating only one entry in the file.

With BSSF, each bit of the signature is stored in a separate file. With a signature size F, the signatures are distributed over F files, instead of one file as in the SSF approach. This is especially useful if we have large signatures. In this case, we only have to search the files corresponding to the bit fields where the query signature has a "1". This can reduce the search time considerably. However, each update implies updating up to F files, which is very expensive. So, even if retrieval cost has been shown to be much smaller for BSSF, the update cost is much higher, 100-1000 times higher is not

uncommon [7]. Thus, BSSF based approaches are most appropriate for relatively static data.

4 Object and Index Management in TOODB

We start with a description of how OID indexing and version management can be done in a TOODB. This brief outline is not based on any existing system, but the design is close enough to make it possible to integrate into current OODBs if desired.

4.1 Temporal OID Indexing

In a traditional OODB, the OIDX is usually realized as a hash file or a B^+-tree, with ODs as entries, and using the OID as the key. In a TOODB, we have more than one version of some of the objects, and we need to be able to access current as well as old versions efficiently. Our approach to indexing is to have *one* index structure, containing all ODs, current as well as previous versions.

In this paper, we assume one OD for each object version, stored in a B^+-tree. We include the commit time *TIME* in the OD, and use the concatenation of OID and time, *OID*||*TIME*, as the index key. By doing this, ODs for a particular OID will be clustered together in the leaf nodes, sorted on commit time. As a result, search for the current version of a particular OID as well as retrieval of a particular time interval for an OID can be done efficiently. When a new object is *created*, i.e., a new OID allocated, its OD is appended to the index tree as is done in the case of the Monotonic B^+-tree [5]. This operation is very efficient. However, when an object is *updated*, the OD for the new version *has to be inserted into the tree.*

While several efficient multiversion access methods exist, e.g., TSB-tree [8], they are not suitable for our purpose, because they provide more flexibility than needed, e.g., combined key range and time range search, at an increased cost. We will never have search for a (consecutive) range of OIDs, OID search will always be for *perfect match*, and most of them are assumed to be to the current version. It should be noted that our OIDX is inefficient for many typical temporal queries. As a result, additional secondary indexes can be needed, of which TSB-tree is a good candidate. However, *the OIDX is still needed*, to support navigational queries, one of the main features of OODBs compared to relational database systems.

4.2 Temporal Object Management

In a TOODB, it is usually assumed that most accesses will be to the current versions of the objects in the database. To keep these accesses as efficient as possible, and to benefit from object clustering, the database is partitioned. Current objects reside in one partition, and the previous versions in the other partition, in the *historical database*. When an object is updated in a TOODB, the previous version is first moved to the historical database, before the new version is stored in-place in the current database.

We assume that clustering is not maintained for historical data, so that all objects going historical, i.e., being moved because they are replaced by a new version, can be

written sequentially, something which reduces update costs considerably. The OIDX is updated *every time an object is updated*.

Not all the data in a TOODB is temporal, for some of the objects, we are only interested in the current version. To improve efficiency, the system can be made aware of this. In this way, some of the data can be defined as non-temporal. Old versions of these are not kept, and objects can be updated in-place as in a traditional OODB, and the costly OIDX update is not needed when the object is modified. This is an important point: using an OODB which efficiently supports temporal data management, should not reduce the performance of applications that do not utilize these features.

5 Storing Signatures in the OIDX

The signature can be stored together with the mapping information (and timestamp in the case of TOODBs) in the OD in the OIDX. Perfect match queries can use the signatures to reduce the number of objects that have to be retrieved, only the candidate objects, with matching signatures, need to be retrieved.

Optimal signature size is very dependent of data and query types. In some cases, we can manage with a very small signature, in other cases, for example in the case of text documents, we want a much larger signature size. It is therefore desirable to be able to use different signature sizes for different kind of data, and as a result, we should provide different signature sizes.

The maintenance of object signatures implies computational overhead, and is not always required or desired. Whether to maintain signatures or not, can for example be decided on a per class basis.

6 Analytical Model

Due to space constraints, we can only present a brief overview of the cost model used. For a more detailed description, we refer to [10]. The purpose of this paper is to show the benefits of using signatures in the OIDX, so we will restrict this analysis to attribute matches, using the superimposed coding technique.

Our cost model focus on disk access costs, as this is the most significant cost factor. In our disk model, we distinguish between random and sequential accesses. With random access, the time to read or write a page is denoted T_P, with sequential access, the time to read or write a page is T_S. All our calculations are based on a page size of 8 KB.

The system modeled in this paper, is a page server OODB, with temporal extensions as described in the previous sections. To reduce disk I/O, the most recently used index and object pages are kept in a *page buffer* of size M_{pbuf}. OIDX pages will in general have low locality, and to increase the probability of finding a certain OD needed for a mapping from OID to physical address, the most recently used ODs are kept in a separate OD cache of size M_{ocache}, containing N_{ocache} ODs. The OODB has a total of M bytes available for buffering. Thus, when we talk about the memory size M, we only consider the part of main memory used for buffering, not main memory used for the

objects, the program itself, other programs, the operating system, etc. The main memory size M is the sum of the size of the page buffer and the OD cache, $M = M_{\text{pbuf}} + M_{\text{ocache}}$.

With increasing amounts of main memory available, buffer characteristics are very important. To reflect this, our cost model includes buffer performance as well, in order to calculate the hit rates of the OD cache and the page buffer. Our buffer model is an extension of the Bhide, Dan and Dias LRU buffer model [2]. An important feature of the BDD model, which makes it more powerful than some other models, is that it can be used with *non-uniform access distributions*. The derivation of the BDD model in [2] also includes an equation to calculate the number N_d of distinct objects out of a total of N access objects, given a particular access distribution. We denote this equation $N_{distinct}(N_d, N)$. The buffer hit probability of an object page is denoted P_{buf_opage}. Note that even if index and object pages share the same page buffer, the buffer hit probability is different for index and object pages. We do not consider the cost of log operations, because the logging is done to separate disks, and the cost is independent of the other costs.

To analyze the use of signatures in the OIDX, we need a cost model that includes:

1. OIDX update and lookup costs.
2. Object storage and retrieval costs.

The OIDX lookup and update costs can be calculated with our previously published cost model [10]. The only modification done to this cost model is that signatures are stored in the object descriptors (ODs). As a consequence, the OD size varies with different signature sizes. In practice, a signature in an OD will be stored as a number of bytes, and for this reason, we only consider signature sizes that are multiples of 8 bits.

The average OIDX lookup cost, i.e., the average time to retrieve the OD of an object, is denoted $T_{\text{oidx_lookup}}$, and the average time to do an update is $T_{\text{oidx_update}}$. Not all objects are temporal, and in our model, we denote the fraction of the operations done on temporal objects as P_{temporal}. For the non-temporal objects, if signatures are not maintained, the OIDX is only updated when the objects are created.

6.1 Object Storage and Retrieval Cost Model

One or more objects are stored on each disk page. To reduce the object retrieval cost, objects are often placed on disk pages in a way that makes it likely that more than one of the objects on a page that is read, will be needed in the near future. This is called clustering. In our model, we define the clustering factor C as the fraction of an object page that is relevant, i.e., if there are N_{o_page} objects on each page, and n of them will be used, $C = \frac{n}{N_{o_page}}$. If $N_{o_page} < 1.0$, i.e., the average object size is larger than one disk page, we define $C = 1.0$.

Read Objects. We model the database read accesses as 1) *ordinary object accesses*, assumed to benefit from the database clustering, and 2) *perfect match queries*, which can benefit from signatures. We assume the perfect match queries to be a fraction P_{qm} of the read accesses, and that P_A is the fraction of queried objects that are actual drops. Assuming a clustering factor of C, the average object retrieval cost, excluding OIDX

lookup, is $T'_{readobj} = \frac{1}{CN_{o_page}} T_{readpage}$, where the average cost of reading one page from the database, is $T_{readpage} = (1 - P_{buf_opage})T_P$. When reading object pages during signature based queries, we must assume we can not benefit from clustering, because we retrieve only a very small amount of the total number of objects. In that case, one page must be read for every object that is retrieved, $T''_{readobj} = T_{readpage}$. The average object retrieval cost, employing signatures, is:

$$T_{readobj} = T_{\text{oidx_lookup}} + (1 - P_{qm})T'_{readobj} + P_{qm}(P_A T''_{readobj} + (1 - P_A)F_d T''_{readobj})$$

which means that of the P_{qm} that are queries for perfect match, we only need to read the object page in the case of actual or false drops. The false drop probability when a signature with F bits is generated from D attributes is denoted $F_d = (\frac{1}{2})^m$, where $m = \frac{F \ln 2}{D}$.

Update Objects. Updating can be done in-place, with write-ahead logging (but note that in the case of temporal objects, these are moved to the historical partition before the new current version is written). In that case, a transaction can commit after its log records have been written to disk. Modified pages are not written back immediately, this is done lazily in the background as a part of the buffer replacement and checkpointing. Thus, a page may be modified several times before it is written back.

Update costs will be dependent of the checkpoint interval. The checkpoint interval is defined to be the number of objects that can be written between two checkpoints. The number of written objects, N_{CP}, includes created as well as updated objects. $N_{CR} = P_{new} N_{CP}$ of the written objects are creations of new objects, and $(N_{CP} - N_{CR})$ of the written objects are updates of existing objects.

The number of distinct updated objects during one checkpoint period is $N_{DU} = N_{distinct}(N_{CP} - N_{CR}, N_{obj})$. The average number of times each object is updated is $N_U = \frac{N_{CP} - N_{CR}}{N_{DU}}$. During one checkpoint interval, the number of pages in the current partition of the database that is affected is $N_P = \frac{N_{DU}}{N_{o_page} C}$ This means that during one checkpoint interval, new versions must be inserted into N_P pages. CN_{o_page} objects on each page have been updated, and each of them have been updated an average of N_U times. For each of these pages, we need to write $P_{\text{temporal}} N_U C N_{o_page}$ objects to the historical partition (this includes objects from the page and objects that were not installed into the page before they went historical), install the new current version to the page, and write it back. This will be done in batch, to reduce disk arm movement, and benefits from sequential writing of the historical objects. For each of the object updates, the OIDX must be updated as well. In the case of a non-temporal OODB, we do not need to write previous versions to the historical partition, and the OIDX needs only to be updated if signatures are to be maintained.

When new objects are created, an index update is needed. When creating a new object, a new page will, on average, be allocated for every N_{o_page} object creation. When a new page is allocated, installation read is not needed. The average object update cost, excluding OIDX update cost:

$$T_{write_obj} = T_{\text{oidx_update}} + \frac{N_P T_S (P_{\text{temporal}} N_U C N_{o_page}) + N_P T_P + \frac{N_{CR}}{N_{o_page}} T_P}{N_{CP}}$$

Note that objects larger than one disk page will usually be partitioned, and each object is indexed by a separate large object index tree. This has the advantage that when a new version is created, only the modified parts need to be written back. An example of how this can be done is the EXODUS large objects [3].

7 Performance

We have now derived the cost functions necessary to calculate the average object storage and retrieval costs, with different system parameters and access patterns, and with and without the use of signatures. We will in this section study how different values of these parameters affect the access costs. Optimal parameter values are dependent of the mix of updates and lookups, and they should be studied together. If we denote the probability that an operation is a write, as P_{write}, the average access cost is the weighted cost of average object read and write operations:

$$T_{\text{access}} = (1 - P_{\text{write}})T_{\text{lookup}} + P_{\text{write}}T_{\text{update}}$$

Our goal in this study is to minimize T_{access}. We measure the gain from the optimization as: Gain $= 100\left(\frac{(T^{nonopt}_{\text{access}} - T^{opt}_{\text{access}})}{T^{opt}_{\text{access}}}\right)$ where $T^{nonopt}_{\text{access}}$ is the cost if not using signatures, and T^{opt}_{access} is the cost using signatures.

Access Model The access pattern affects storage and retrieval costs directly and indirectly, through the buffer hit probabilities. The access pattern is one of the parameters in our model, and is modeled through the independent reference model. Accesses to objects in the database system are assumed to be random, but skewed (some objects are more often accessed than others), and the objects in the database can be logically partitioned into a number of partitions, where the size and access probability of each partition can be different. With β_i denoting the relative size of a partition, and α_i denoting the percentage of accesses going to that partition, we can summarize the four access patterns used in this paper:

Set	β_0	β_1	β_2	α_0	α_1	α_2
3P1	0.01	0.19	0.80	0.64	0.16	0.20
3P2	0.001	0.049	0.95	0.80	0.19	0.01
2P8020	0.20	0.80	-	0.80	0.20	-
2P9505	0.05	0.95	-	0.95	0.05	-

In the first partitioning set, we have three partitions. It is an extensions of the 80/20 model, but with the 20% hot spot partition further divided into a 1% hot spot area and a 19% less hot area. The second partitioning set, 3P2 resembles the access pattern close to what we expect it to be in future TOODBs. The two other sets in this analysis have each two partitions, with hot spot areas of 5% and 20%.

Parameter	Value	Parameter	Value	Parameter	Value
M	100 MB	S_{obj}	128	C	0.3
M_{ocache}	0.2 M	N_{objver}	100 mill.	D	1
N_{CP}	$0.8N_{\text{ocache}}$	S_{od}	$32 + \lceil \frac{F}{8} \rceil$	P_{new}	0.2
P_A	0.001	P_{write}	0.2	P_{qm}	0.4
P_{temporal}	0.8				

Table 1. Default parameters.

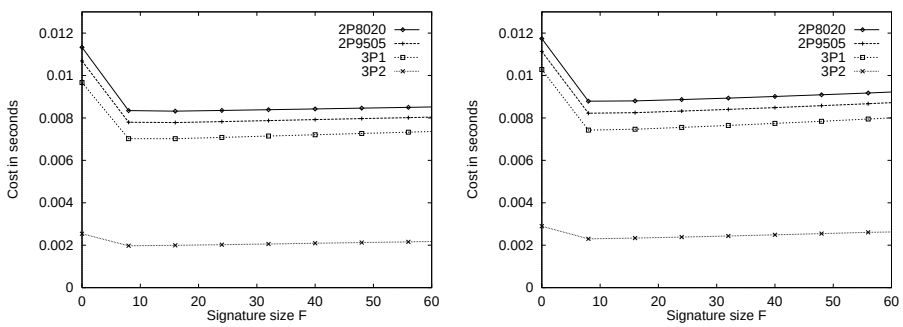


Fig. 1. Cost versus signature size for different access patterns. Non-temporal OODB to the left, temporal OODB to the right.

Parameters We consider a database in a stable condition, with a total of N_{objver} objects versions (and hence, N_{objver} ODs in the OIDX). Note that with the OIDX described in Section 4.1, OIDX performance is not dependent of the number of existing versions of an object, only the total number of versions in the database.

Unless otherwise noted, results and numbers in the next sections are based on calculations using default parameters, as summarized in Table 1, and access pattern according to partitioning set 3P1.

With the default parameters, the studied database has a size of 13 GB. The OIDX has a size of 3 GB in the case of a non-temporal OODB, and 5 GB in the case of a temporal OODB (not counting the extra storage needed to store the signatures). Note that the OIDX size is smaller in a non-temporal OODB, because in a non-temporal OODB, we do not have to store timestamps, and we have no inserts into the index tree, only append-only. In that case, we can get a very good space utilization [4]. When we have inserts into the OIDX, as in the case of a temporal OODB, we get a space utilization in the OIDX that is less than 0.67.

7.1 Optimal Signature Size

Choosing the signature size is a tradeoff. A larger signature can reduce the read costs, but will also increase the OIDX size and OIDX access costs. Figure 1 illustrates this for different access patterns. In this case, a value of $F = 8$ seems to be optimal. This

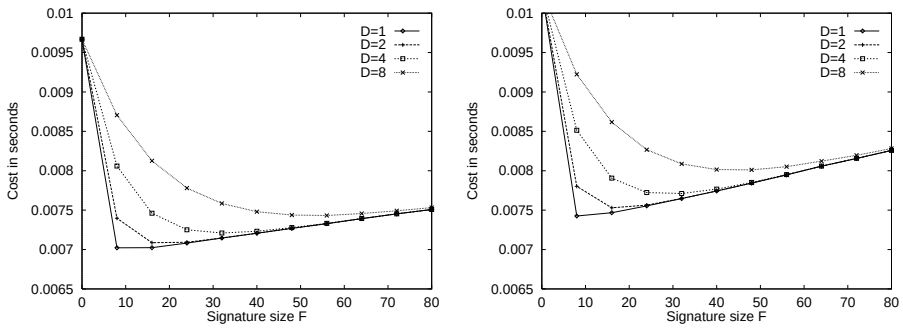


Fig. 2. Cost versus signature size for different values of D. Non-temporal OODB to the left, temporal OODB to the right.

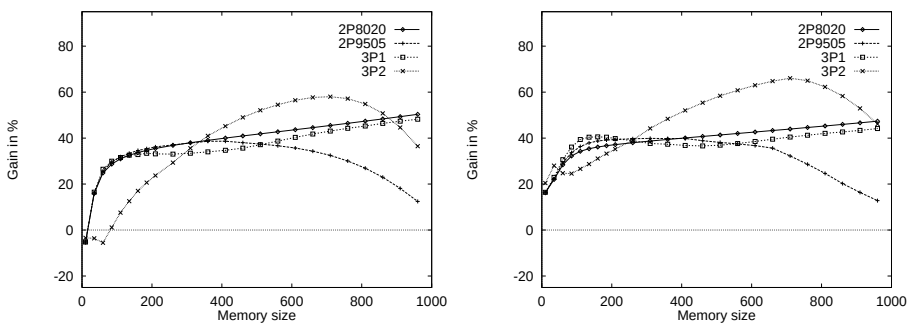


Fig. 3. Gain from using signatures versus memory size, for different access patterns. Non-temporal OODB to the left, temporal OODB to the right.

is quite small, and gives a higher false drop probability than accepted in text retrieval applications. The reason why such a small signature is optimal in our context, is that the size of objects is small enough to make object retrieval and match less costly than a document (large object) retrieval and subsequent search for matching word(s), as is the case in text retrieval applications.

The signature size is dependent of D, the number of attributes contributing to the signature. This is illustrated in Fig. 2. With an increasing value of D, the optimal signature size increases as well. In our context, a value of D larger than one, means that more than one attribute contributes to the signature, so that queries on more than one attribute can be performed later.

In the rest of this study, we will use F=8 when using the default parameters, and use F=16, 32 and 48 for D=2, 4, and 8, respectively.

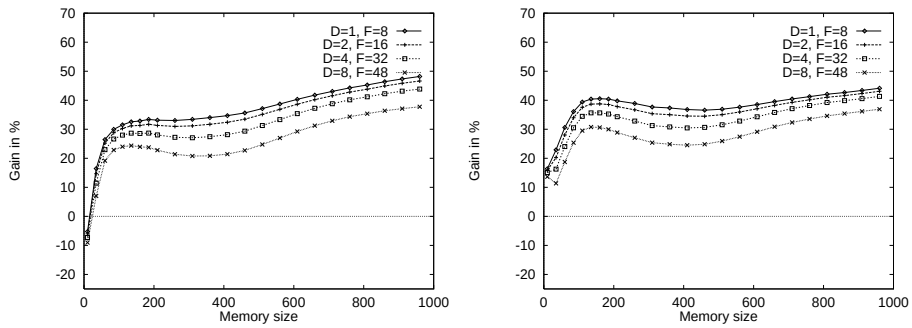


Fig. 4. Gain from using signatures, versus memory size, for different values of D. Non-temporal OODB to the left, temporal OODB to the right.

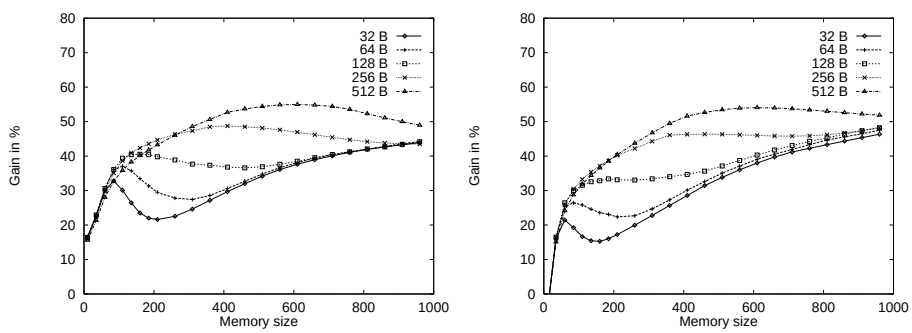


Fig. 5. Gain from using signatures, versus memory size, for different average object sizes. Non-temporal OODB to the left, temporal OODB to the right.

7.2 Gain From Using Signatures

Figure 3 shows the gain from using signatures, with different access patterns. Using signatures is beneficial for all access patterns, except when only a very limited amount of memory is available.

Figure 4 shows the gain from using signatures, for different values of D. The gain decreases with increasing value of D.

7.3 The Effect of the Average Object Size

We have chosen 128 as the default average object size. It might be objected that this value is too large, but Fig. 5 shows that even with smaller object sizes, using signatures will be beneficial. The figure also shows how the gain increases with increasing object size.

7.4 The Effect of P_A and P_{qm}

The value of P_{qm} is the fraction of the read queries that can benefit from signatures. Figure 6 illustrates the gain with different values of P_{qm}. As can be expected, a small

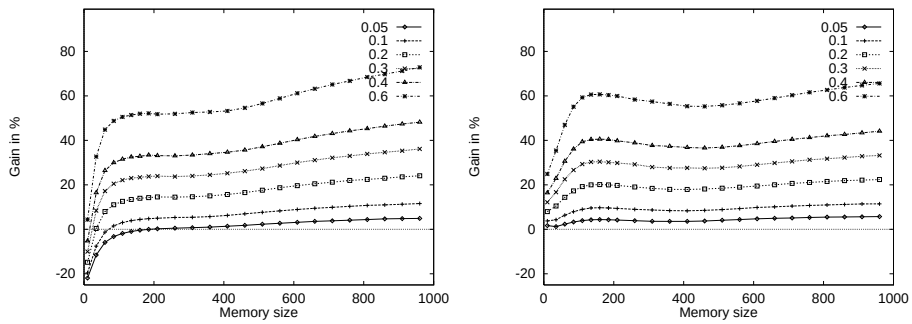


Fig. 6. Gain from using signatures, versus memory size, for different values of P_{qm}. Non-temporal OODB to the left, temporal OODB to the right.

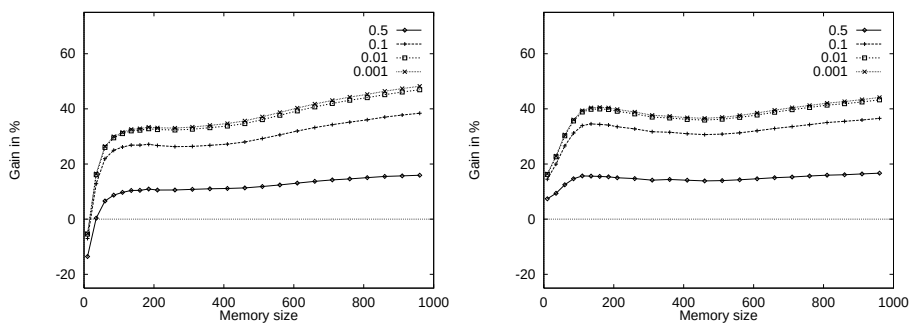


Fig. 7. Gain from using signatures, versus memory size, for different values of P_A. Non-temporal OODB to the left, temporal OODB to the right.

value of P_{qm} results in negative gain in the case of non-temporal OODBs, i.e., in this case, storing and maintaining signatures in the OIDX reduces the average performance.

The value of P_A is the fraction of queries objects that are actual drops, the selectivity of the query. Only if the value of P_A is sufficiently low, will we be able to benefit from using signatures. Figure 7 shows that signatures will be beneficial even with a relatively large value of P_A.

8 Conclusions

We have in this paper described how signatures can be stored in the OIDX. As the OD is accessed on every object access in any case, there is no extra signature retrieval cost. In a traditional, non-versioned OODBs, maintaining signatures means that the OIDX needs to be updated every time an object is updated, but as the analysis shows, it will in most cases pay back, as less objects need to be retrieved.

Storing signatures in the OIDX is even more attractive for TOODBs. In TOODBs, the OIDX will have to be updated on every object update anyway, so in that case, the extra cost associated with signature maintenance is very low.

As showed in the analysis, substantial gain can be achieved by storing the signature in the OIDX. We have done the analysis with different system parameters, access patterns, and query patterns, and in most cases, storing the object signatures in the OIDX is beneficial. The typical gain is from 20 to 40%. Interesting to note is that the optimal signature size can be quite small.

The example analyzed in this paper is quite simple. A query for perfect match has a low complexity, and there is only limited room for improvement. The real benefit is available in queries where the signatures can be used to reduce the amount of data to be processed at subsequent stages of the query, resulting in larger amounts of data that can be processed in main memory. This can speed up query processing several orders of magnitude.

Another interesting topic for further research is using signatures in the context of bitemporal TOODBs.

Acknowledgments

The author would like to thank Olav Sandstå for proofreading, and Kjell Bratbergsengen, who first introduced him to signatures.

References

1. E. Bertino and F. Marinaro. An evaluation of text access methods. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, 1989. Vol.II: Software Track*, 1989.
2. A. K. Bhide, A. Dan, and D. M. Dias. A simple analysis of the LRU buffer policy and its relationship to buffer warm-up transient. In *Proceedings of the Ninth International Conference on Data Engineering*, 1993.
3. M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the 12th VLDB Conference, Kyoto, Japan, August 1986*, 1986.
4. A. Eickler, C. A. Gerlhof, and D. Kossmann. Performance evaluation of OID mapping techniques. In *Proceedings of the 21st VLDB Conference*, 1995.
5. R. Elmasri, G. T. J. Wuu, and V. Kouramajian. The time index and the monotonic B^+-tree. In A. U. Tansel, J. Clifford, S. K. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal databases: theory, design and implementation*. The Benjamin/Cummings Publishing Company, Inc., 1993.
6. C. Faloutsos and R. Chan. Fast text access methods for optical and large magnetic disks: Designs and performance comparison. In *Proceedings of the 14th VLDB Conference*, 1988.
7. Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evaluation of signature files as set access facilities in OODBs. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993.
8. D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD*, 1989.
9. K. Nørvåg and K. Bratbergsengen. Log-only temporal object storage. In *Proceedings of the 8th International Workshop on Database and Expert Systems Applications, DEXA'97*, 1997.
10. K. Nørvåg and K. Bratbergsengen. Optimizing OID indexing cost in temporal object-oriented database systems. In *Proceedings of the 5th International Conference on Foundations of Data Organization, FODO'98*, 1998.

Appendix G

Validation of the Index Buffer Model

In this appendix we validate the general hierarchical index buffer model presented in the paper in Appendix C, and subsequently used in the analytical models in the papers presented in Appendix D, E, and F. Our buffer models are based on the BDD LRU buffer model [16], described in Section 14.4. The assumptions behind the BDD model include independent accesses. However, in our index buffer model, accesses are only partially independent. The validation has been done by comparing simulation results with the analytical model. The simulations have been performed with different index sizes, buffer sizes, index page fanouts, and access patterns.

G.1 The Index Buffer Simulator

The simulator itself is quite simple. It maintains an LRU chain of index pages currently resident in the buffer. The simulator can operate with either a *traverse* or a *no traverse* strategy.

Using a traverse strategy means that a complete traversal is needed from root to leaf for every leaf node access. In some index implementations, however, the leaf page would be self describing. When doing a search in the tree, we would first check if the relevant leaf page is already in the buffer. Only if the leaf page is not resident, we would need to traverse the tree. We call this a *no traverse strategy*.

The equations derived in the paper in Appendix C, assumed a hot buffer. Therefore, we warm up the buffer by doing a large number of requests, before we start measuring the buffer hit probability.

All simulation results in this section are results from simulations with an index tree with 200000 leaf pages.[1] We have also performed simulations using larger index trees, but with the same qualitative result. The index buffer simulations have been done using two partitioning sets, summarized in Table G.1. Partitioning set 1 is used as the default set.

[1] 200000 leaf pages are, with a leaf page size of 1024 entries, sufficient to index approximately 200 million objects.

Partition Size, Set 1	Partition Size, Set 2	Partition Access Probability
$\beta_0^0 = 0.01$	$\beta_0^0 = 0.001$	$\alpha_0^0 = 0.64$
$\beta_1^0 = 0.19$	$\beta_1^0 = 0.199$	$\alpha_1^0 = 0.16$
$\beta_2^0 = 0.80$	$\beta_2^0 = 0.80$	$\alpha_2^0 = 0.20$

Table G.1: Partition sets used in the index buffer validation.

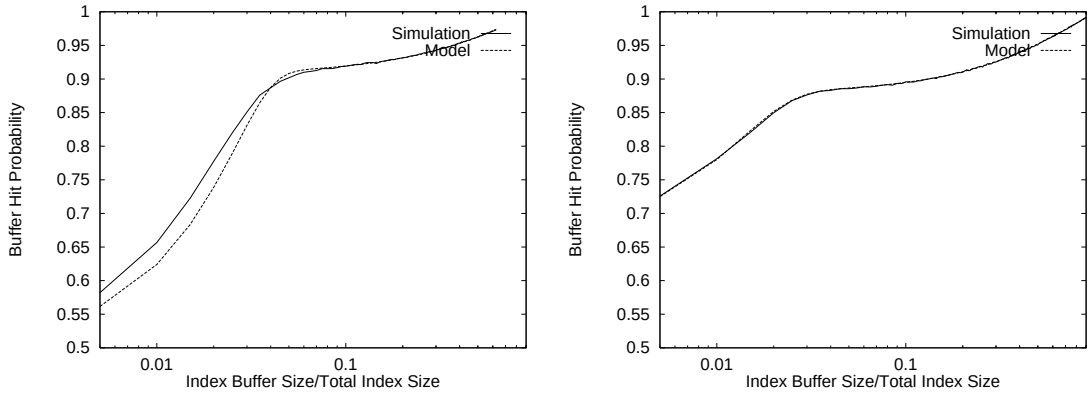


Figure G.1: Overall buffer hit probability with different buffer sizes, fanout F=64 to the left, and fanout F=2048 to the right.

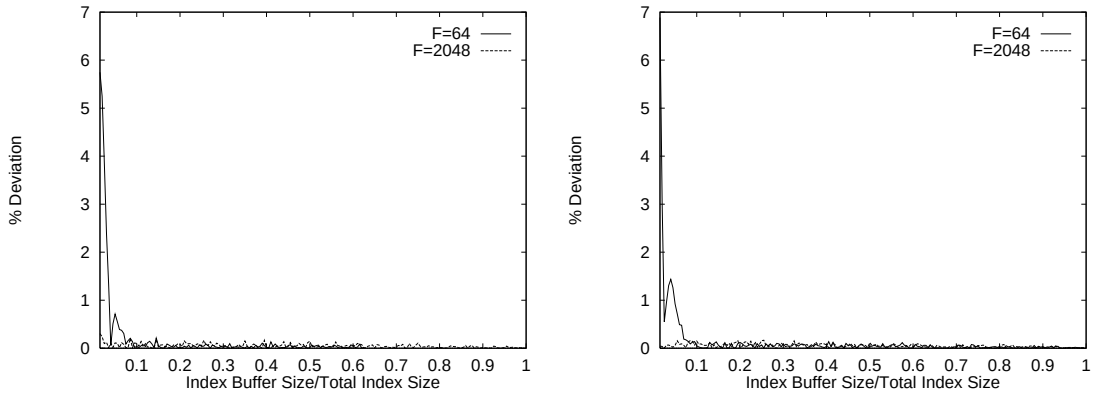


Figure G.2: Relative deviation between buffer hit rate in analytical model and simulations, partitioning set 1 to the left, partitioning set 2 to the right.

G.2 Results

Figure G.1 shows the overall buffer hit probability for indexes with fanout F=64 and F=2048 (note that the number of leaf pages is the same for both indexes). We see clearly how the buffer hit performance increases close to the point where a new complete index tree level has space in the buffer. After that point, it rises more slowly, until it reaches the point when most of the next level fits completely in the buffer. In the figure, we also see how close the model is to the simulation results (we have used a logarithmic scale for the buffer size axis to emphasize deviations, without logarithmic scale, the curves would be overlapping).

Figure G.2 shows the relative deviation between estimated and simulated buffer hit rate. The deviation is very small. The left hand plot is with partitioning set 1, the right hand plot is with partitioning set 2.

The results presented until this point have been achieved using a traverse strategy. When simulating a no traverse strategy, we are satisfied if the leaf page requested is in the buffer. With the traverse strategy, we assume that the leaf pages in the buffer do not contain enough information to be accessed

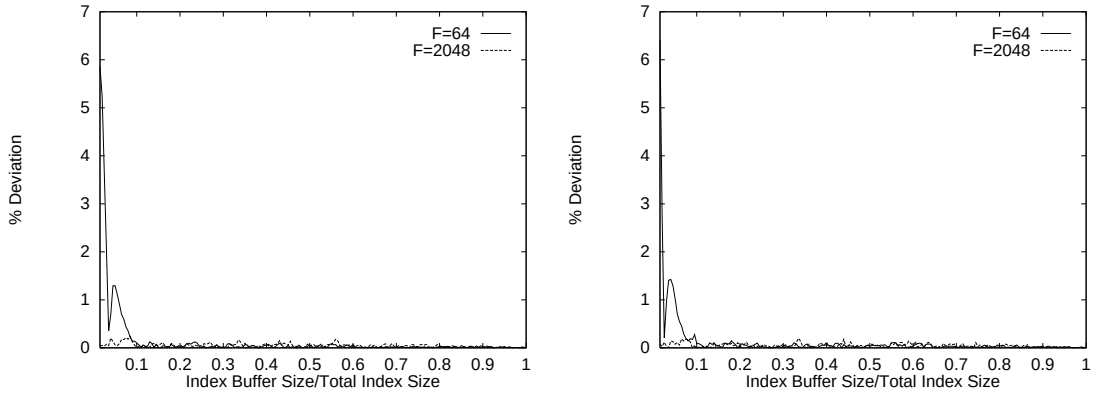


Figure G.3: Relative deviation between buffer hit rate in analytical model and simulations when the no traverse strategy is used, partitioning set 1 to the left, partitioning set 2 to the right.

directly. In this case, we traverse the tree from the root node to leaf page, and do a buffer allocation and replacement each time we ask for a node that is not resident in the buffer. An access to a node resident in buffer, makes the node move to the front of the buffer chain.

To study the impact of a no traverse strategy on the buffer hit probability, we have compared the simulations done with the no traverse strategy, and compared the results with the results of the analytical model, which assumed complete traversal. In Figure G.3, we have plotted the relative deviation between estimated and simulated buffer hit rate in the case of a no traverse strategy. This shows that the deviation is small in this case as well. We also measured the deviation with other partitioning sets, and got the same result.

G.3 Related Work

Concurrently with our work and publication of the index buffer model, a similar index buffer model based on the BDD model has been presented by Leutenegger and Lopez in [125]. Their index buffer model was done in the context of buffering of R-tree nodes, but can also be applied to B-trees. The main difference between our model and the one presented by Leutenegger and Lopez, is that their models only consider a uniform access pattern.

G.4 Conclusions

We have now compared the index buffer model with simulations using different database and workload parameters, and have shown that the model has an acceptable accuracy.

Appendix H

Abbreviations

This appendix contains abbreviations which are defined in this thesis, and abbreviations that we do not expect every reader of the thesis to be acquainted with. The chapter and section references are to the chapter or section where the abbreviations are defined or described.

- **2PC:** Two-phase commit
- **2PL:** Two-phase locking
- **API:** Application program interface
- **BDD model:** The Bhide, Dan and Dias LRU buffer model, Section 14.4
- **BSSF:** Bit-sliced signature file, Section 7.1.2
- **CAD:** Computer aided design
- **CDO:** Class descriptor object, Section 6.2.1
- **CID:** Class identifier, Section 8.1.2
- **DIB:** Device information block, Section 13.1.1
- **CONTID:** Container identifier, Section 8.1.1
- **CONTIDX:** Container index, Section 8.4
- **CVOIDX:** Current version OIDX, Section 8.4
- **DBMS:** Database management system
- **GIS:** Geographical information system, Section 1.1
- **HVOIDX:** Historical version subindex, Section 8.4
- **IPU-ODBMS:** In-place update ODBMS, Chapter 14
- **LFS:** Log-structured file system, Section 5.4.2
- **LHAM:** Log-structured history data access method, Section 5.4.3

- **LO-ODBMS:** Log-only ODBMS, Chapter 14
- **MMDBMS:** Main-memory database management system
- **NavigDesc:** Navigational descriptor, Section 6.2.1 and Section 8.5.2
- **OD:** Object descriptor, Section 6.2 and Section 8.1.2
- **ODMG:** Object Data Management Group (previously Object Database Management Group), Section 2.2
- **OID:** Object identifier, Section 3.1
- **OIDX:** Object identifier index, Section 3.1
- **ODBMS:** Object database management system, Section 2.1
- **OODB:** Object-oriented database, see ODBMS
- **ORDBMS:** Object-relational database management system, Section 2.3.3
- **PACS:** Picture archiving and communications systems, Section 1.1
- **PCache:** Persistent cache, Chapter 9
- **PCST:** PCache status table, Chapter 9.3
- **RDBMS:** Relational database management system
- **RSOT:** Resident small object table, Section 13.2.2
- **SGID:** Server group identifier, Section 8.1.1
- **SMP:** Symmetric multiprocessor
- **SOD:** Subobject descriptor, Section 6.2 and Section 8.5.2
- **SOH:** Special object handler, Section 10.1
- **SSDB:** Scientific and statistical databases, Section 1.1
- **SSF:** Sequential signature file, Section 7.1.2
- **SST:** Segment status table, Chapter 5.1
- **TOIDX:** Temporal object identifier index
- **TID:** Transaction identifier, Section 12.1
- **TTCT:** TID/timestamp/counter table, Section 12.2.2
- **USN:** Unique serial number, Section 8.1.1
- **VDT:** Volume device table, Section 12.12
- **VTOIDX:** The Vagabond Temporal OIDX, Section 8.4
- **WAL:** Write-ahead logging

References

[1] S. Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory*, 1997.

[2] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Siméon. Querying documents in object databases. *International Journal on Digital Libraries*, 1(1), 1997.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1), 1997.

[4] I. Ahn and R. Snodgrass. Partitioned storage for temporal databases. *Information Systems*, 13(4), 1988.

[5] M. M. Astrahan et al. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2), 1976.

[6] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, 1992.

[7] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, 1992.

[8] Barry & Associates, Inc. *Object Storage Fact Book 4.0: Object DBMSs*. 1998.

[9] L. Bellatreche, K. Karlapalem, and A. Simonet. Horizontal class partitioning in object-oriented databases. In *Proceedings of the 8th International Conference on Database and Expert Systems Applications, DEXA'97*, 1997.

[10] V. Benzaken, C. Delobel, and G. Harrus. Clustering strategies in O_2: An overview. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, 1992.

[11] C. A. v. d. Berg. *Dynamic Query Processing in a Parallel Object-Oriented Database System*. PhD thesis, University of Twente, 1994.

[12] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company Inc., 1987.

[13] E. Bertino and C. Guglielmina. Optimization of object-oriented queries using path indices. In *Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*. IEEE Computer Society Press, 1992.

[14] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), 1989.

[15] E. Bertino and F. Marinaro. An evaluation of text access methods. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, 1989. Vol.II: Software Track*, 1989.

[16] A. K. Bhide, A. Dan, and D. M. Dias. A simple analysis of the LRU buffer policy and its relationship to buffer warm-up transient. In *Proceedings of the Ninth International Conference on Data Engineering*, 1993.

[17] A. Biliris and E. Panagos. The BeSS object storage manager: Architecture overview. *SIGMOD Record*, 25(3), 1996.

[18] S. M. Blackburn and R. B. Stanton. Multicomputer object stores: the multicomputer Texas experiment. In S. Nettles and R. Conner, editors, *Seventh International Workshop on Persistent Object Systems*, 1996.

[19] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceeding of the Winter 1995 Usenix Conference*, 1995.

[20] J. Blakeley. Open OODB: Architecture and query processing overview. In A. Dogac, M. T. Özsu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems — Proceedings of the NATO Advanced Study Institute on Object-Oriented Database Systems, Izmir, Kusadasi, Turkey, August 6-16, 1993*. Springer-Verlag, 1994.

[21] L. Boeszoermenyi, J. Eder, and C. Weich. PPOST: A parallel database in main-memory. In *Proceedings of the 5th International Conference on Database and Expert Systems Applications, DEXA'94*, 1994.

[22] L. Boeszoermenyi, K.-H. Eder, and C. Weich. PPOST - Persistent Parallel Object STore. In *Proceedings of the International Conference on Massively Parallel Processing Applications and Development, MPP'94*, 1994.

[23] L. Boeszoermenyi and C. Weich. Simple and efficient transactions for a distributed object store. In *Proceedings of the Ninth International Workshop on Database and Expert Systems Applications, DEXA'98*, 1998.

[24] M. H. Böhlen. Temporal database system implementation. *SIGMOD Record*, 24(4), 1995.

[25] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in temporal databases. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*. Morgan Kaufmann, 1996.

[26] P. Boncz and M. L. Kersten. Monet: An impressionist sketch of an advanced database system. In *BIWIT'95*, 1995.

[27] P. Boncz, F. Kwakkel, and M. Kersten. High performance support for OO traversals in Monet. In *BNCOD 96*, 1996.

[28] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–23, 1990.

[29] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of the 10th International Conference on VLDB*, 1984.

[30] K. Bratbergsengen and K. Nørvåg. Improved and optimized partitioning techniques in database query processing. In *Proceedings of the Fifteenth British National Conference on Databases, BNCOD15, Lecture Notes in Computer Science (LNCS), vol. 1271*. Springer-Verlag, 1997.

[31] K. Bratbergsengen and K. Nørvåg. Optimizing hash partitioning in relational algebra. In *Proceedings of Norsk Informatikkonferanse 1996*, Alta, Norway, November 1996.

[32] K. Bratbergsengen, O. Risnes, and T. Amble. ASTRAL: A structured and unified approach to data base design and manipulation. In *IFIP TC-2 Working Conference on Data Base Architectures*, 1979.

[33] K. Bratbergsengen and T. Stålhane. Feature analysis of Astral. In J. W. Schmidt and M. L. Brodie, editors, *Relational Database Systems: Analysis and Comparison*. Springer-Verlag, 1983.

[34] K. Buehler and L. McKee, editors. *The OpenGIS Guide — Introduction to Interoperable Geoprocessing and the OpenGIS Specification. Third Edition*. Open GIS Consortium, Inc., 1998.

[35] P. Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1997.

[36] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *Proceedings of 5th Conference on Architectural Support for Programming Languages and Operating Systems, October 1992*, 1992.

[37] P. Butterworth, A. Otis, and J. Stein. The GemStone object database management system. *Communications of the ACM*, 34(10), 1991.

[38] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the 12th VLDB Conference*, 1986.

[39] M. J. Carey, D. J. DeWitt, M. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proceedings of the 1994 ACM SIGMOD Conference*, 1994.

[40] M. J. Carey, M. J. Franklin, and M. Zaharioudakis. Fine-grained sharing in a page server OODBMS. In *Proceedings of the 1994 ACM SIGMOD*, 1994.

[41] M. J. Carey et al. O-O, what have they done to DB2? In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99)*, 1999.

[42] E. Casais, M. Ranft, B. Schiefer, D. Theobald, and W. Zimmer. OBST — An overview. Technical report, Forschungszentrum Informatik (FZI), 1992.

[43] R. Cattell, editor. *The Object Database Standard: ODMG-93. Release 2.0*. Morgan Kaufmann, 1997.

[44] RD45 - Basic concepts. Technical report, European Organization for Nuclear Research, 1996.

[45] Object databases and mass storage systems: The prognosis. Technical Report CERN/LHCC 96-17, LCRB/RD45, European Organization for Nuclear Research, 1996.

[46] Using an object databases and mass storage system for physics analysis. Technical Report CERN/LHCC 97-9, LCB/RD45, European Organization for Nuclear Research, 1997.

[47] Status report of the RD45 project. Technical Report CERN/LHCC 98-11, LCB/RD45, European Organization for Nuclear Research, 1998.

[48] Y.-H. Chen and S. Y. W. Su. Implementation and evaluation of parallel query processing algorithms and data partitioning heuristics in object-oriented databases. *Distributed and Parallel Databases*, 4(2), 1996.

[49] W. Cockshott, M. P. Atkinson, K. Chisholm, P. Bailey, and R. Morrison. Persistent object management systems. *Software — Practice and Experience*, 14(49-71), 1984.

[50] G. P. Copeland and S. Khoshafian. A decomposition storage model. In *Proceedings of SIGMOD International Conference on Management of Data*, 1985.

[51] B. David, L. Raynal, G. Schorter, and V. Mansart. GeO_2: Why objects in a geographical DB. In *Third International Symposium, SSD '93, Singapore, June 1993*, 1993.

[52] L. Daynes and M. Atkinson. Main-memory management to support orthogonal persistence for Java. In *The Second International Workshop on Persistence and Java(tm) (PJW2)*, 1997.

[53] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10), 1991.

[54] D. J. DeWitt, P. Futtersack, D. Maier, and F. Velez. A study of three alternative workstation-server architectures for object-oriented database systems. In *Proceedings of the 20th VLDB Conference*, 1990.

[55] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-server paradise. In *Proceedings of the 20th VLDB Conference*, 1994.

[56] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *Proceedings of the 17th International Conference on Very Large Data Bases*, 1991.

[57] D. J. DeWitt, J. F. Naughton, J. C. Shafer, and S. Venkataraman. Parallelising OODBMS traversals: A performance evaluation. *VLDB Journal*, 5(1), 1996.

[58] A. Dogac, C. Ozkan, B. Arpinar, T. Okay, and C. Evrendilek. METU object-oriented DBMS. In A. Dogac, M. T. Özsu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems — Proceedings of the NATO Advanced Study Institute on Object-Oriented Database Systems, Izmir, Kusadasi, Turkey, August 6-16, 1993*. Springer-Verlag, 1994.

[59] P. Drew, R. King, and S. Hudson. The performance and utility of the Cactis algorithms. In *Proceedings of the 16th VLDB Conference*, 1990.

[60] C. E. Dyreson and R. T. Snodgrass. Timestamp semantics and representation. *Information Systems*, 18(3), 1993.

[61] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4), 1984.

[62] A. Eickler, C. A. Gerlhof, and D. Kossmann. Performance evaluation of OID mapping techniques. In *Proceedings of the 21st VLDB Conference*, 1995.

[63] K. Elhardt and R. Bayer. A database cache for high performance and fast restart in database systems. *ACM Transactions on Database Systems*, 9(4), 1984.

[64] R. Elmasri, G. T. J. Wuu, and V. Kouramajian. The time index and the monotonic B^+-tree. In A. U. Tansel, J. Clifford, S. K. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal databases: theory, design and implementation*. The Benjamin/Cummings Publishing Company, Inc., 1993.

[65] C. Faloutsos. Access methods for text. *ACM Computer Surveys*, 17(1), 1985.

[66] C. Faloutsos and R. Chan. Fast text access methods for optical and large magnetic disks: Designs and performance comparison. In *Proceedings of the 14th VLDB Conference*, 1988.

[67] L. Fegaras and R. Elmasri. A temporal object query language. In *Proceedings of the Fifth International Workshop on Temporal Representation and Reasoning*, 1998.

[68] D. Fishman, D. Beech, H. Cate, C. Chow, T. Connors, J. Davis, N. Derrett, C. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. Shan. Iris: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1), Jan. 1987.

[69] D. A. Ford and J. Myllymaki. A log-structured organization for tertiary storage. In *The 12th International Conference on Data Engineering*, 1996.

[70] K. Fosse. Objektbuffer i objektorienterte databasesystem (in Norwegian). Master's thesis, NTNU, 1998.

[71] N. Gehani. The Ode object-oriented database management system: An overview. In A. Dogac, M. T. Özsu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems — Proceedings of the NATO Advanced Study Institute on Object-Oriented Database Systems, Izmir, Kusadasi, Turkey, August 6-16, 1993*. Springer-Verlag, 1994.

[72] J. A. Gendrano, B. C. Huang, J. M. Rodrigue, B. Moon, and R. T. Snodgrass. Parallel algorithms for computing temporal aggregates. In *Proceedings of the Fifteenth International Conference on Data Engineering*, 1999.

[73] C. A. Gerlhof, A. Kemper, and G. Moerkotte. On the cost of monitoring and reorganization of object bases for clustering. *Sigmod Record*, 25(3), 1996.

[74] M. Gesmann. Mapping a parallel complex-object DBMS to operating system processes. In *Proceedings of Euro-Par'96 - Parallel Processing, Workshop: Parallel and Distributed Database Systems*, 1996.

[75] M. Gesmann, A. Grasnickel, T. Härder, C. Hübel, W. Käfer, B. Mitschang, and H. Schöning. PRIMA - a database system supporting dynamically defined composite objects. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, 1992.

[76] S. Ghandeharizadeh, D. Wilhite, K. Lin, and X.Zhao. Object placement in parallel object-oriented database systems. In *Proceedings of the 10th International Conference on Data Engineering*, 1994.

[77] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[78] R. J. Green, A. C. Baird, and J. C. Davies. Designing a fast on-line backup system for a log-structured file system. *Digital Technical Journal*, 8(2), 1996.

[79] R. Grossman and X. Qin. Ptool: A scalable persistent object manager. In *Proceedings of the 1994 ACM SIGMOD*, 1994.

[80] D. S. Group. Overview of the Amadeus project (Amadeus v2.0). Technical Report TCD-CS-92-01, Dept. of Computer Science, Trinity College Dublin, Feb. 1992.

[81] O. Gruber, L. Amsaleg, L. Daynès, and P. Valduriez. Eos, an environment for object-based systems. In *25th Hawaii International Conference on System Sciences*, 1992.

[82] O. Gruber and P. Valduriez. Object management in parallel database servers. In P. Valduriez, editor, *Parallel processing and Data Management*. Chapman & Hall, 1992.

[83] H. Gunadhi and A. Segev. Efficient indexing methods for temporal relations. *IEEE Transactions on Knowledge and Data Engineering*, 5(3), 1993.

[84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*, June 1984.

[85] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[86] R. F. Haddleton. *Parallel Set Operations in Complex Object-Oriented Queries*. PhD thesis, University of Virginia, 1998.

[87] N. W. Hiroyuki Kitagawa and Y. Ishikawa. Design and evaluation of signature file organization incorporating vertical and horizontal decomposition schemes. In *Proceedings of the 7th International Conference on Database and Expert Systems Applications, DEXA'96*, 1996.

[88] U. Hohenstein, V. Pleßer, and R. Heller. Evaluating the performance of object-oriented database systems by means of a concrete application. In R. R. Wagner, editor, *Proceedings of the 8th International Workshop on Database and Expert Systems Applications, DEXA'97*, Toulouse, France, 1997. IEEE Computer Society Press.

[89] M. F. Hornick and S. B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems*, 5(1), 1987.

[90] T. Härder, H. Schöning, and A. Sikeler. Parallel query evaluation: A new approach to complex object processing. *Data Engineering Bulletin*, 12(1), 1989.

[91] D. Hulse and A. Dearle. A log-structured persistent store. In *Proceedings of the 19th Australasian Computer Science Conference*, 1996.

[92] D. Hulse, A. Dearle, and A. Howells. Lumberjack: A log-structured persistent object store. In *Proceedings of the Eighth International Workshop on Persistent Object Systems (POS8)*, 1998.

[93] S.-O. Hvasshovd, Øystein Torbjørnsen, S.-E. Bratsberg, and P. Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21st VLDB Conference*, 1995.

[94] S. J. Hyun and S. Y. Su. Parallel query processing strategies for object-oriented temporal databases. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, 1996.

[95] H. Ishikawa. *Object-Oriented Database System: Design and Implementation for Advanced Applications*. Springer-Verlag, 1993.

[96] Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evaluation of signature files as set access facilities in OODBs. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993.

[97] ITASCA. Distributed object database system. Technical summary release 2.3. IBEX Computing, 1995.

[98] B. R. Iyer and D. Wilhite. Data compression support in databases. In *Proceedings of the 20th VLDB Conference*, 1994.

[99] H. V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dalí: A High Performance Main Memory Storage Manager. In *Proceedings of the 20th VLDB Conference*, pages 48–59, Santiago, Chile, 1994.

[100] C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1), 1999.

[101] C. S. Jensen (Editor). A consensus glossary of temporal database concepts. *SIGMOD Record*, 23(1), 1994.

[102] J. E. Johnson and W. A. Laing. Overview of the Spiralog file system. *Digital Technical Journal*, 8(2), 1996.

[103] K. Karlapalem, S. B. Navathe, and M. M. A. Morsi. Issues in distribution design of object-oriented databases. In M. T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management — Edited collection of papers presented at the International Workshop on Distributed Object Management 1992*. Morgan Kaufmann, 1994.

[104] P. Åke Larson and G. Graefe. Memory management during run generation in external sorting. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, 1998.

[105] J. Kempe, W. Kowarschick, W. Kiessling, R. Hitzelberger, and F. Dutkowski. The OCAD benchmark for object-oriented database systems. Technical Report FR-1995-004, Bayerisches Forschungszentrum fur Wissensbasierte Systeme (FORWISS), Munich, Germany, 1995.

[106] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases: Design, realization and evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 6(4), 1994.

[107] A. Kemper and D. Kossmann. Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis. In *Proceedings IEEE International Conference on Data Engineering*, pages 155–162, 1993.

[108] A. Kemper and D. Kossmann. Dual-buffering strategies in object bases. In *Proceedings of the 20th VLDB Conference*, 1994.

[109] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proceedings of the 16th VLDB Conference*, 1990.

[110] A. Kemper and G. Moerkotte. Query optimization in object bases: Exploiting relational techniques. In *Query Processing for Advanced Applications*, pages 63–94. Morgan-Kaufmann, 1993.

[111] K.-C. Kim. Parallelism in object-oriented query processing. In *IEEE Sixth International Conference on Data Engineering*, 1990.

[112] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), Mar. 1990.

[113] H. Kitagawa and K. Fukushima. Composite bit-sliced signature file: An efficient access method for set-valued object retrieval. In *Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications (CODAS)*, 1996.

[114] H. Kitagawa, Y. Fukushima, Y. Ishikawa, and N. Ohbo. Estimation of false drops in set-valued object retrieval with signature files. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, 1993.

[115] W. Klas, P. Fankhauser, P. Muth, and T. R. E. J. Neuhold. Database integration using the open object-oriented database system VODAK. In O. Bukhres and A. K. Elmagarmid, editors, *Object Oriented Multidatabase Systems: A Solution for Advanced Applications*. Prentice Hall, 1996.

[116] N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *Proceedings of IEEE 11th International Conference on Data Engineering, March, 1995*, 1995.

[117] J. T. Kohl, C. Staelin, and M. Stonebraker. HighLight: Using a log-structured file system for tertiary storage management. In *Proceedings of the USENIX Winter 1993 Conference*, 1993.

[118] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10), 1991.

[119] D. E. Langworthy and S. B. Zdonik. Extensibility and asynchrony in the brown-object storage system. In V. Kumar, editor, *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice Hall, 1996.

[120] D. L. Lee, Y. M. Kim, and G. Patel. Efficient signature file methods for text retrieval. *IEEE Transactions on Knowledge and Data Engineering*, 7(3), 1995.

[121] C. Leung and D. Taniar. Parallel query processing in object-oriented database systems. *Australian Computer Science Communications*, 17(2), 1995.

[122] T. Y. C. Leung and R. R. Muntz. Query processing for temporal databases. In *Proceedings of ICDE 1990*, 1990.

[123] T. Y. C. Leung and R. R. Muntz. Generalized data stream indexing and temporal query processing. In *Proceedings of RIDE-TQP'92, Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, 1992.

[124] T. Y. C. Leung and R. R. Muntz. Temporal query processing and optimization in multiprocessor database machines. In *Proceedings of the 18th VLDB Conference*, 1992.

[125] S. Leutenegger and M. Lopez. The effect of buffering on the performance of R-trees. In *Proceedings of the 1998 International Conference on Data Engineering*, 1998.

[126] D. Lieuwen, D. DeWitt, and M. Mehta. Parallel pointer-based join techniques for object-oriented databases. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, 1993.

[127] D. F. Lieuwen. *Optimizing and Parallelizing Loops in Object-Oriented Database Programming Languages*. PhD thesis, University of Wisconsin-Madison, 1992.

[128] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, U. M. R. Gruber, A. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *SIGMOD '96*, 1996.

[129] B. Liskov, A. Adya, M. Castro, and Q. Zondervan. Type-safe heterogeneous sharing can be fast. In *Seventh International Workshop on Persistent Object Systems, May 1996*, 1996.

[130] B. Liskov, M. Day, and L. Shrira. Distributed object management in Thor. In M. T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management — Edited collection of papers presented at the International Workshop on Distibuted Object Management 1992*. Morgan Kaufmann, 1994.

[131] G. M. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer. Extensions to Starburst: Objects, types, functions, and rules. *Communications of the ACM*, 34(10), 1991.

[132] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD*, 1989.

[133] D. Maier and J. Stein. Development and implementation of an object-oriented DBMS. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Systems*. Morgan Kaufmann, 1990.

[134] Matisse 2.3. technical overview, 1995.

[135] F. Matthes, G. Schröder, and J. Schmidt. Tycoon: A scalable and interoperable persistent system environment. In M. Atkinson, editor, *Fully Integrated Data Environments*. Springer-Verlag, 1995.

[136] M. L. McAuliffe. *Storage Management Methods for Object Database Systems*. PhD thesis, University of Wisconsin-Madison, 1997.

[137] C. McHugh and V. Cahill. Eiffel**: An implementation of Eiffel on Amadeus, a persistent, distributed applications support environment. Technical Report TCD-CS-93-36, Trinity College, Dublin, Ireland, 1993.

[138] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, , and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3), 1997.

[139] W. J. McIver, Jr. and R. King. Self-adaptive, on-line reclustering of complex object data. In *Proceedings of the 1994 ACM SIGMOD*, 1994.

[140] J. E. B. Moss. Addressing large distributed collections of persistent objects: The Mneme project's approach. Technical report, University of Massachusetts, Amherst, 1989.

[141] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. Technical report, University of Massachusetts, Amherst, 1991.

[142] J. E. B. Moss and S. Sinofsky. Managing persistent data with Mneme: Designing a reliable, shared object interface. In K. Dittrich, editor, *Advances in Object-Oriented Database Systems — 2nd International Workshop on Object-Oriented Database Systems*. Springer-Verlag, 1988.

[143] P. Muth, P. O'Neil, A. Pick, and G. Weikum. Design, implementation, and performance of the LHAM log-structured history data access method. In *Proceedings of the 24th VLDB Conference*, 1998.

[144] J. M. Neefe, D. Roselli, A. Costello, R. Wang, and T. Anderson. Improving the performance of log structured file systems with adaptive methods. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, 1997.

[145] R. S. Nikhil and M. L. Heytens. Exploiting parallelism in the implementation of Agna, a persistent programming system. In *IEEE 7th International Conference on Data Engineering*, 1991.

[146] S. Nittel and K. R. Dittrich. A storage server for the efficient support of complex objects. In R. Conner and S. Nettles, editors, *Seventh International Workshop on Persistent Object Systems*, 1996.

[147] K. Nørvåg. Efficient use of signatures in object-oriented database systems. In *Proceedings of Advances in Databases and Information Systems, ADBIS'99*, 1999.

[148] K. Nørvåg. The Persistent Cache: Improving OID indexing in temporal object-oriented database systems. In *Proceedings of the 25th VLDB Conference*, 1999.

[149] K. Nørvåg. An approach to high-performance scalable temporal object storage. In *Proceedings of the Workshop on High Performance Object Databases (HiPOD'2000)*, 2000.

[150] K. Nørvåg. A comparative study of log-only and in-place update based temporal object database systems. In *Proceedings of the Ninth International Conference on Information and Knowledge Management (CIKM 2000) (To be published)*, 2000.

[151] K. Nørvåg. Design issues in transaction-time temporal object database systems. In *Proceedings of ADBIS-DASFAA'2000*, 2000.

[152] K. Nørvåg. Main-memory management in temporal object database systems. In *Proceedings of ADBIS-DASFAA'2000*, 2000.

[153] K. Nørvåg. A performance evaluation of log-only temporal object database systems. In *Proceedings of the 12th International Conference on Scientific and Statistical Database Management (SSDBM 2000)*, 2000.

[154] K. Nørvåg. A study of signature caching in parallel object database systems (submitted for publication), 2000.

[155] K. Nørvåg. The Vagabond temporal OID index: An index structure for OID indexing in temporal object database systems. In *Proceedings of the 2000 International Database Engineering and Applications Symposium (IDEAS)*, 2000.

[156] K. Nørvåg. Fine-granularity signature caching in object database systems. Technical Report IDI 6/2000, Norwegian University of Science and Technology, 2000.

[157] K. Nørvåg. The Vagabond parallel temporal object-oriented database system: Versatile support for future applications. In *Proceedings of Norsk Informatikkonferanse 1999*, Trondheim, Norway, November 1999.

[158] K. Nørvåg and K. Bratbergsengen. Log-only temporal object storage. In *Proceedings of the 8th International Workshop on Database and Expert Systems Applications, DEXA'97*, 1997.

[159] K. Nørvåg and K. Bratbergsengen. Write optimized object-oriented database systems. In *Proceedings of the XVII International Conference of the Chilean Computer Science Society, SCCC'97*, 1997.

[160] K. Nørvåg and K. Bratbergsengen. An analytical study of object identifier indexing. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications, DEXA'98*, 1998.

[161] K. Nørvåg and K. Bratbergsengen. Optimizing OID indexing cost in temporal object-oriented database systems. In *Proceedings of the 5th International Conference on Foundations of Data Organization, FODO'98*, 1998.

[162] K. Nørvåg and K. Bratbergsengen. Optimal object descriptor caching in temporal object database systems. In K. Tanaka, S. Ghandeharizadeh, and Y. Kambayashi, editors, *Information Organization and Databases*. Kluwer Academic Publishers (To be published), 2000.

[163] K. Nørvåg and K. Bratbergsengen. Aggregate and grouping functions in object-oriented databases. In *Proceedings of the XVI International Conference of the Chilean Computer Science Society, SCCC'97*, November 1996.

[164] K. Nørvåg, O. Sandstå, and K. Bratbergsengen. Concurrency control in distributed object-oriented database systems. In *Advances in Databases and Information Systems - ADBIS'97 (Also published in Springer-Verlag electronic Workshops in Computing)*, St. Petersburg, Russia, September 1997.

[165] O_2 System Administration Guide, Release 4.6, April 1996.

[166] O_2 ODMG C++ Binding Guide, Release 4.6, April 1996.

[167] Objectivity technical overview, version 5, 1999.

[168] ODB-II: technical overview, 1998.

[169] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4), 1996.

[170] P. O'Neil and G. Weikum. A log-structured history data access method. In *Proceedings of the 5th International Workshop on High Performance Transaction Systems, Asilomar, Sept. 1993*, 1993.

[171] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query processing in the ObjectStore database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1992.

[172] G. Panagopoulos and C. Faloutsos. Bit-sliced signature files for very large databases on a parallel machine architecture. In *Proceedings of EDBT'94. 4th International Conference on Extending Database Technology*, 1994.

[173] J. Patel, J. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. Hall, K. Ramasamy, R. Lueder, C. Ellman, J. Kupsch, S. Guo, D. DeWitt, and J. Naughton. Building a scalable geo-spatial DBMS: technology, implementation, and evaluation. In *Proceedings of the 1997 SIGMOD Conference*, 1997.

[174] J. L. Pfaltz, R. F. Haddleton, and J. C. French. Scalable, parallel, scientific databases. In *Proceedings of SSDBM'98*, 1998.

[175] T. Printezis, M. Atkinson, L. Daynes, S. Spence, , and P. Bailey. The design of a new persistent object store for PJama. In *The Second International Workshop on Persistence and Java(tm) (PJW2)*, 1997.

[176] R. Rastogi, S. Seshadri, P. Bohannon, D. Leinbaugh, A. Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databases. In *Proceedings of the 23rd VLDB Conference*, 1997.

[177] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, 1991.

[178] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2), 1999.

[179] H. Samet. *Applications of Spatial Data Structures*. Addison-Wesley Publishing Company Inc., 1990.

[180] H.-J. Schek, H.-B. Paul, M. H. Scholl, and G. Weikum. The DASDBS project: Objectives, experiences and future prospects. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[181] J. W. Schmidt, M. Mall, and W. H. Dotzek. Feature analysis of the PASCAL/R relational system. In J. W. Schmidt and M. L. Brodie, editors, *Relational Database Systems: Analysis and Comparison*. Springer-Verlag, 1983.

[182] M. Seltzer. Transaction support in a log-structured file system. In *Proceedings of the Ninth International Conference on Data Engineering*, 1990.

[183] M. Seltzer. *File System Performance and Transaction Support*. PhD thesis, University of California at Berkeley, 1992.

[184] M. Seltzer and M. Stonebraker. Transaction support in read optimized and write optimized file systems. In *Proceedings of the 16th VLDB Conference*, 1990.

[185] M. Selzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Conference*, 1993.

[186] E. Shekita and M. Zwilling. Cricket: A mapped, persistent object store. Technical Report CS-TR-90-956, University of Wisconsin-Madison, 1990.

[187] E. J. Shekita and M. J. Carey. Performance enhancement through replication in an object-oriented DBMS. In *Proceedings of the 1989 ACM SIGMOD*, 1989.

[188] E. J. Shekita and M. J. Carey. A performance evaluation of pointer-based joins. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 1990.

[189] V. Singhal, S. Kakkad, and P. Wilson. Texas: An efficient, portable persistent store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, 1992.

[190] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Transitioning temporal support in TSQL2 to SQL3. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases: Research and Practice*. Springer-Verlag, 1998.

[191] R. T. Snodgrass (ed.), I. Ahn, G. Ariav, D. S. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. G. Kulkarni, T. Y. C. Leung, N. A. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada. *The TSQL2 temporal query language*. Kluwer Academic, 1995.

[192] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient evaluation of the valid-time natural join. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 282–292. IEEE Computer Society, 1994.

[193] B. Sreenath and S. Seshadri. The hcC-tree: An efficient index structure for object oriented databases. In *Proceedings of the 20th VLDB Conference*, 1994.

[194] A. Steiner. *A Generalisation Approach to Temporal Data Models and their Implementations*. PhD thesis, Swiss Federal Institute of Technology, 1998.

[195] M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 13th VLDB Conference*, 1987.

[196] M. Stonebraker. *Readings in Database Systems (2nd edition)*. Morgan Kaufmann, 1994.

[197] M. Stonebraker and G. Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10), 1991.

[198] M. Stonebraker and M. Olson. Large object support in POSTGRES. In *Proceedings of IEEE 9th International Conference on Data Engineering*, 1993.

[199] M. Stonebraker and L. A. Rowe. The design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, 1986.

[200] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), Mar. 1990.

[201] S. Y. Su, R. Jawadi, P. Cherukuri, Q. Li, and R. Nartey. OSAM*.KBMS/P: A parallel, active, object-oriented knowledge base server. Technical Report TR94-031, University of Florida, 1994.

[202] T. Suzuki and H. Kitagawa. Development and performance analysis of a temporal persistent object store POST/C++. In *Proceedings of the 7th Australasian Database Conference*, 1996.

[203] B. Tangney, A. Condon, V. Cahill, and N. Harris. Requirements for parallel programming in object-oriented distributed systems. *The Computer Journal*, 37(6):499–508, Aug. 1994.

[204] Techra query language self instruction guide, 1987.

[205] W. Teeuw, C. Rich, M. Scholl, and H. Blanken. An evaluation of physical disk I/Os for complex object processing. In *Proceedings of the 9th International Conference on Data Engineering*, 1993.

[206] P. Tiberio and P. Zezula. Selecting signature files for specific applications. In *Proceedings of 5th Annual European Computer Conference on Advanced Computer Technology, Reliable Systems and Applications, CompEuro '91*, 1991.

[207] M. Tsangaris and J. Naughton. A stochastic approach to clustering in object bases. In *Proceedings of SIGMOD'91*, 1991.

[208] M. Tsangaris and J. Naughton. On the performance of object clustering techniques. In *Proceedings of SIGMOD'92*, 1992.

[209] S. Venkataraman. *Global Memory Management for Multi-Server Database Systems*. PhD thesis, University of Wisconsin-Madison, 1996.

[210] S. Venkataraman, M. Livny, and J. Naughton. Impact of data placement on memory management for multi-server OODBMS. In *International Conference on Data Engineering*, 1995.

[211] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, December 1993*, 1993.

[212] C. Whitaker, J. S. Bayley, and R. D. W. Widdowson. Design of the server for the Spiralog file system. *Digital Technical Journal*, 8(2), 1996.

[213] S. J. White. *Pointer Swizzling Techniques for Object-Oriented Database Systems*. PhD thesis, University of Wisconsin-Madison, 1994.

[214] S. J. White and D. J. DeWitt. QuickStore: A high performance mapped object store. In *Proceedings of the 1994 ACM SIGMOD*, 1994.

[215] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris architecture and implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[216] Y. Wu, S. Jajodia, and X. S. Wang. Temporal database bibliography update. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases: Research and Practice*. Springer-Verlag, 1998.

[217] J. Yoo, M. Kim, Y. Lee, and B. Im. Performance evaluation of dynamic signature file methods. In *Proceedings of Computer Software and Applications Conference, COMPSAC'95*, 1995.

[218] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.

[219] M. Özsu and J. Blakeley. Query processing in object-oriented database systems. In W. Kim, editor, *Modern Database Management — Object-Oriented and Multidatabase Technologies*. Addison-Wesley/ACM Press, 1994.

[220] T. Zurek. Optimisation of partitioned temporal joins. In *Proceedings of the Fifteenth British National Conference on Databases, BNCOD15, Lecture Notes in Computer Science (LNCS), vol. 1271*. Springer-Verlag, 1997.