Write Optimized Object-Oriented Database Systems

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

In a database system, read operations are much more common than write operations, and consequently, database systems have been read optimized. As the size of main memory increases, more of the database read requests will be satisfied from the buffer system, and the amount of disk write operations relative to disk read operations will increase. This calls for a focus on write optimized *database systems. In this paper, we present solutions to this problem. We describe in detail the data structures and algorithms needed to realize a write optimized object-oriented database system in the context of Vagabond, an OODB currently being implemented at our department. In Vagabond, focus has been to provide support for applications which have earlier used file systems because of the limited data bandwidth in current database systems, typical examples are super computing applications and geographical information systems*

1. Introduction

Based on current available technology, some interesting observations can be done:

- Disk is cheap.
- Memory is cheap.
- CPU speed is increasing very fast.

However: the increase in disk speed is much lower than the memory and CPU speed, which results in an increasing secondary memory access bottleneck. This is not a new situation, minimizing the effects of this bottleneck has been the main motivation behind most database related research. However, the advent of very large main memory buffers, makes it necessary to revise previous work and solutions.

In a database system, most accesses to data are read operations. Traditional database systems are designed to work in an environment where memory is expensive, and hence, buffer space quite limited. As a result, they have been *read optimized*. However, the increasing buffer size, means that more and more of the read requests can be satisfied from the buffer system. The result is that performance become limited by the the systems write throughput. This calls for a focus on *write optimized* database systems. Another aspect that gives this issue increased importance, is database systems which also needs high performance in terms of data bandwidth, and not only in transaction throughput (although these points are related). This is especially important for new emerging application areas, like super computing applications and geographical information systems, which have earlier used file systems.

In the rest of the paper, we will show how we can increase write throughput in an object-oriented database system (OODB). This is done by employing techniques from write-optimized file systems, mainly *log-structured file systems* (LFS) [11]. Some important differences between common file system requirements and database requirements complicates this work. The most important is data granularity. Objects are much more lightweight than files. The overhead acceptable for finding a file in a file system, is not acceptable to find an object. Second, the number of objects in an object-oriented database system will usually be much larger than the number of files in a file system. While it is possible to have most of the file directory cached, this will not be possible for the object directory. Finally, access patterns are radically different.

A database system must also provide ACID properties. In file systems, data are often not immediately written to disk (unless flushing is done explicitly) when a file is closed.

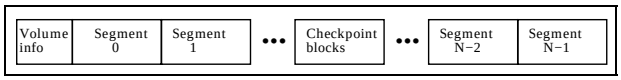

Figure 1. Disk volume structure.

The reason is that commonly, created files are just temporary files, that will be deleted shortly after. In this case, disk write would be wasted. While this is acceptable in a file system environment, it is of course not acceptable for a database storage system. When a transaction is committed, the data should really be durable.

1.1. Outline of the Paper

The rest of this paper is organized as follows. Section 2 presents the log-only approach, and its advantages. Section 3 gives an overview of the architecture of the Vagabond OODB. Section 4 describes objects in Vagabond, and Section 5 describes how to index objects in the log. Section 6 describes the physical storage structures, and Section 7 how to do access data in the database, transaction management, and recovery. Section 8 gives an overview of related work, and in Section 9 we conclude the paper and indicate topics for future work.

2. The Log-Only Approach

Increasing the effective disk bandwidth can be done in three ways: 1) by reducing seek time, 2) by reducing the amount of time spent in rotational delay, and 3) parallel I/O. In current database systems, the first two are partially achieved by the use of write ahead logging (WAL), which defers the non-sequential writing. However, sooner or later, the data has to be written to the database. This involves the writing of lots of small objects, almost always one access for each individual object. Our solution to this problem, is *to eliminate the database completely*, and use a *log-only* approach. The log is written contiguously to the disk, in a no-overwrite way, in large blocks. This is similar to the approach used in *log structured file systems* (LFS) [11]. We will now explain briefly how data are written to the log, and delve into more details regarding algorithms and data structures in the following sections.

2.1. Log Writing

With a log-only approach, data as well as metadata are written contiguously to the log. To be able to retrieve data written to the log, an index, the OID-index (OIDX), is used to map from from logical object identifier (OID) to physical location on disk. This index structure is interleaved with the

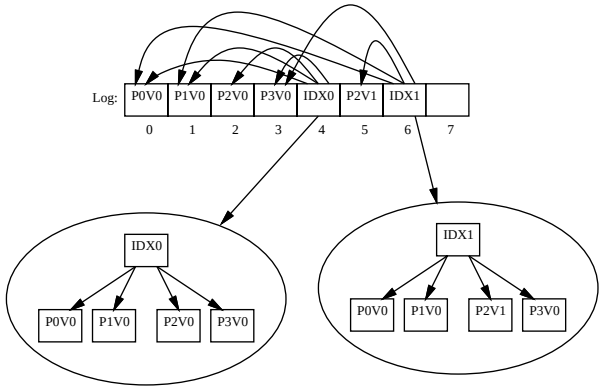

Figure 2. Data and index in a log-only OODB.

objects in the log. Already written data are never modified, new versions of the objects and the index are just appended to the log. Writes are always done sequentially, and with a large write block size, preferably up to 1MB. This is done by writing many objects and index entries, possibly from many transactions, in one write operation.

Logically, the log is an infinite length resource, but the physical disk size is, of course, not infinite. We solve this problem by dividing the disk volume into large, equal sized, physical segments, as illustrated on Figure 1.

A segment starts in a *clean state*, e.g., 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 *alive*, it contains valid data. When checkpoint is done, all previous segments in the alive state changes state to *dirty*.

Recovery in a log-only database can be done very fast, since there is no need to redo or undo any data. Only segments that were *alive* when the system crashed needs to be processed. At regular times, a checkpoint operation is performed, and at recovery time we just do an analysis pass from the last known checkpoint to the end of the log (where the crash occurred).

As data are deleted, old segments can be reused. Deleted data will leave behind a lot of partially filled segments, the data in these near empty segments can be collected and moved to a new segment, thus freeing up space in the old segments and making the old segments available for reuse. This process is called *cleaning*.

Figure 2 illustrates how data and index are interleaved in the log. On top of the figure is the log. At time t_0, a transaction allocates four objects or pages. The index block IDX0 is written, and the pages can later be retrieved via this index (which, in general, have more than one level above the leaf

pages). Later, a new transaction modifies one of the objects on page number 2 (whose first version is denoted P2V0). The new version of the page (page P2V1) and a new version of the index (index block IDX1) is written to the log. The two versions of the database are illustrated on the figure, with arrows from the respective index blocks. If we want to keep the old version, e.g., in a temporal database system, we use an index that can index more than one version of an object.

2.2. Log-Only Object Storage

There are two alternative ways to design an OODB based on LFS techniques:

1. Page based [5, 14], and
2. Object based.

Page Based Designs. In these designs, we look at the log as one large persistent address space. When objects are created, they are allocated space from this address space. These pages are written to the log, as illustrated on Figure 2. The objects are referenced by persistent memory address, and are retrieved via the page index 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.

Object Based Designs. The alternative to a page based design, is to to index objects instead of pages in the persistent address space. When an object is modified, only the object (or a delta object) needs to be written to the log. This is especially useful if good clustering is difficult. This is the approach taken in Vagabond, described in detail in the next sections.

Page vs. Object Based Log-Only OODB. The main advantage of the page based approach is ease of implementation. However, it has the same problem as traditional page servers: Even if just 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 write bandwidth. 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, described in the next section.

In the rest of the paper, we write *log-only* as short for a log-only object based design.

2.3. 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 discuss some of the advantages of the approach.

2.3.1. Functional Issues

A log-only approach is particularly applicable for storage technologies where in-data modifications are costly or impossible. Two good examples are write-once optical disks, where in-place update is impossible, and flash memory, where write/erase has to be done blockwise. A log-only approach is also interesting for very large databases where most of the database resides on tape.

Disk access times and bandwidth improve at a much lower rate than main memory, and parallel disk systems, e.g., RAID, are necessary to get high performance. To benefit from RAID technology, the write blocks has to be much larger than those used in traditional systems. In addition, sequential write becomes more important. While in normal systems, sequential write is only about 3-5 times faster than random write, in RAID, sequential write is probably 20 times faster than random write [16]. The advantages of combining LFS and RAID have already been shown in the Sawmill system [13].

Each write of an object to the log creates a new version, which is timestamped. Realizing a temporal database system is easy with our approach. Versioning comes at virtually no extra cost, while in a conventional database system, versioning doubles the amount of data that has to be written (the previous current version has to be moved before the new can be inserted). The sequential writing of the log, with timestamped segments, also makes on-line and incremental backup easy. To take a incremental backup, it is enough to know the last time of backup to know where backup should be started. This also make it possible to do backup at times when the load on the system is low, and temporarily stop it when the load is high.

With in-place update, it is difficult to save storage space by using compression, since compression ratio and storage size might change. However, this is no problem with the log-only approach, objects can change size with no fragmentation problems.

Versioning/timestamping can be exploited by cache coherence algorithms in client-server environments, as is done in BOSS [8]. Similar techniques can also be used in peer-to-peer parallel database systems. It can also help in nomadic computing. By the use of timestamped data, it is easier to update partitioned databases after reconnect.

The log-only approach is particularly attractive in application areas with large objects, e.g., super computing applications. In many super computing 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 indepen-

dently. It is necessary to retrieve and store these chunks efficiently. Until now, only file systems have been able to offer the performance needed. However, there is a demand for some of the services offered by database systems in these areas: access control, concurrency control, and recovery. Performance close to file system performance is necessary for database systems to be applicable.

For high-availability applications, fast crash recovery is needed. The log-only approach is really a refined form of shadow storage. While this has its deficiencies, 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. Only one read pass is needed from the last checkpoint.

The fact that the log-only approach has similarities to earlier shadow page approaches, implies that it might inherit the nasty side of shadowing: after a while, data becomes declustered. Clearly, this can also be the case with our approach. However, we expect that the increased amount of main memory will compensate for the lack of clustering. It is also possible to *recluster* the database when needed. This can be done as a part of the cleaning process. Many systems are write-once systems, and if a large batch is loaded at a time, we can get very efficient clustering. Also, dynamic, adaptive, clustering can be done. Traditional clustering works well as long as the access pattern is static, but if this happens to change, or access pattern is navigational rather than set based, reclustering the database have to be done.

3. Overview of Vagabond

In the rest of the paper, we will describe how a log-only OODB can be implemented. We start with a general overview of the Vagabond architecture.

Vagabond is a system designed for high performance, based on parallel servers. In many organizations it is also desirable to have the data in a distributed system. To satisfy this, we ended up with a hybrid solution: a distributed system, with *server groups*. In this way, objects are clustered on server groups based on locality as is common in conventional distributed OODBs, but one server group can contain more than one computer (a kind of "super server"). Objects to be stored on a server group are declustered on the servers in the group according to some declustering strategy, e.g., hashing.

Similar to another recent project, Shore [2], our architecture is a peer-to-peer architecture. Clients in the system are connected to *one server* running on the same machine. This make it possible for several clients running on the same machine to utilize a common cache. This server is the gateway to the database system. The servers do not have to contain any data (clearly, if all severs did, including those running on office workstations, availability would be a big problem). Thus, even if a node contains no data volume, a server must be running on that node to make it possible for the client to access the OODB. The communication between the client and the server process is through shared memory.

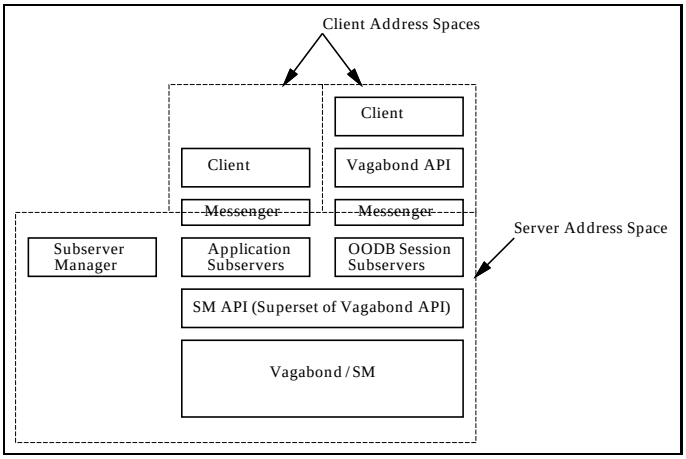

Figure 3. The Vagabond server.

The architecture of the server is shown on Figure 3. A normal client operates against the Vagabond API, which provides the mechanisms to communicate with the server through a shared memory queue, the *messenger*. Each client that connects to the server is allocated one server thread, an *OODB session subserver*. This operates in the server address space on behalf of the client.

Vagabond is extensible, and new subservers can be added to the system. Subservers access the storage manager (SM) through the SM API. The interesting point here, is that the SM API is a superset of the Vagabond API, which the clients operate against. This feature makes it possible 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 on Figure 3.

Instead of using a page buffer as is common in other systems, we use an *object buffer*, which is more efficient when data are not guaranteed to be well clustered.

4. Objects In Vagabond

In our storage system, all objects smaller than a certain threshold , e.g., 64 KB, are written as one contiguous object (not segmented into pages as is done in other systems). Objects larger than this threshold are segmented into *subobjects*, and a *large object index* on the subobjects is maintained. There are several reasons for doing it this way:

- Writing one very large object should not block all other transactions during that time.

OID
Physical location
Create timestamp
Object size
Class tag
Delta object?
Versioned object?
Compressed object?
Large object?

Figure 4. Object descriptor.

- A segmented object is useful later, when only parts of the object is to be read. This also simplifies insert/modify operations to large objects, only inserted/modified parts and the new object index table have to be written.

- Parts of the object can reside on different physical devices, possibly on different levels in the storage hierarchy.

5. Object Indexing

An object in an OODB is uniquely identified by an object identifier (OID). OIDs can be *physical*, or *logical*. If the former is used, the disk block where the object resides is given directly from the OID, if the latter is used, it is necessary to use an index to convert from logical OID to physical location. In our system, logical OID is necessary, since the objects are never written back to the same place. This might seem like an disadvantage, but even though a physical OID has a potential performance benefit, it also has a major drawback: relocation and migration of objects are difficult. Therefore, logical OIDs are generally the preferred choice.

5.1. Object Descriptors

Object descriptors (OD) contains administrative information for each object version. Its main function is to provide the information needed to map from logical OID to physical location. The ODs are stored both in the OID index, and together with the objects in the segment. They are stored in the segments to help identifying objects during cleaning, and as a kind of write ahead logging of index entries. An OD contains the following information, summarized on Figure 4.

Physical location is the location in secondary or tertiary memory. If the physical location is NULL, this means that the object is deleted (but previous versions might exist), or that it is not yet created (timestamp is NULL). If the object is a large object, the location is the location of the (root of) the subobject index of the object.

Create timestamp is the (logical) commit time of the transaction creating this version.

Versioned object is set to true if this is an object where we want to keep old versions when it is modified or deleted. This is decided for each object at object creation time, but can be changed later (although this is, in general, not a good idea).

Often, only small parts of an object is changed when a new version is written. In this case, much can be gained if only the changes are written. This is especially the case if an object is a hot spot. A version which only contains the changes from last version, is called a *delta object*. The delta object itself can be made at low cost by the use of XOR between the new and the old version, and run-length encoding the result. The disadvantage of delta object is, of course, that previous versions have to be retrieved to reconstruct an object at read time. *Delta object* in the OD is set if this version is a delta object. Delta objects are usually only created if the previous version is already in memory. For large objects, other rules apply.

To further reduce storage space, and disk bandwidth, objects can be compressed before they are written. In many application areas, e.g., statistical and scientific databases, a large number of NULL-fields exists in the records/objects. Without even knowing the structure of the objects, it is easy to run-length encode these objects. Compression/decompression is transparent to applications, and a retrieved object is decompressed before it is delivered. *Compressed object* is set when an object is compressed before it is stored.

Class tag identifies the class the object belongs to. This is necessary to be able to use hierarchical concurrency control techniques, and is also useful for type checking.

Large object is true if the object is *large* (cf. Section 4). Both plain "data" objects and index structures are realized as large objects. Clearly, different object classes and indexes have different needs. The class tag, which primary use is tagging an object with its class, is not needed for indexes, we use it as an index class identifier as well. This makes the system extensible, new index classes can be added to the system as needed.

5.2. OID Index

The number of ODs can be very large, and a fast and efficient index structure is needed. Efficient indexing in a log-only system is not trivial. Consider a tree-like index structure: if we update an index node, this will be written to a new location. The pointer in its parent node becomes invalid. The parent node needs to be updated, and this cascades up to the root. We also have another problem: index

blocks as well as objects may be relocated during cleaning or migration. In this case, structures having pointers to this data needs to be updated. To avoid inefficient structures, and minimize cascading updates, design of the index structure needs careful attention.

Each version of an object has its own OD, and this versioning complicates the index considerably, access to current as well as old versions of objects has to be supported by the index. There are several solutions to the indexing problem:

- Two index structures, one for current ODs, and one for historical ODs.
- One index structure for current ODs, and maintaining a linked list of versions from the version in the index.
- One index structure, with all ODs, current as well previous versions.

If versioning is only used to support temporal databases, most queries will be against the current data. To make access to current version as efficient as possible, one separate index for current data is desirable. The problem with this approach, is that every time a new version is created, we have to update *two* indexes.

A second alternative is to have one index structure for current ODs, and linking old versions to new versions. This has similarities with approach used in Postgres [15]. In Postgres, a link exists from one version of a tuple, to the previous version. This has to serious disadvantages: 1) a lot of disk accesses may be necessary to retrieve an old version, 2) this is not efficient for all kinds of queries, an extra index is needed, and 3) most important, in our system, it would be difficult to move objects during cleaning, since only backward pointers exists (bidirectional pointers would make an efficient write pattern impossible).

The third alternative is to have one index structure, with all ODs, current as well as previous versions. This is necessary if versioning is also used for multiversion concurrency control. In that case, both current and *recent* data will be accessed. From research in multiversion access methods the recent years, we know that it is possible to make such a structure efficient, e.g., by using a time-split B-tree [9]. However, the general available multiversion access methods provides more flexibility than we need, an OID index has some special properties that can be exploited to make it more efficient:

- The keys in the index, the OIDs, are not uniformly distributed over a domain as keys commonly are supposed to be.
- If we assume the unique part of an OID to be an integer, new OIDs will always be assigned monotonic increasing values. There will never be insertions of new key (OID) values between existing keys (OIDs).
- If an object is deleted, the OID will never be reused.
- There will be no key range search (but time range search may be needed), search will always be for perfect match.

The index structure in Vagabond is an ISAM variant, with extensions for versions. To reduce the index update cost, we write only the ODs when a transaction is committed (write ahead logging), and write the index blocks themselves later. If an index block to be updated is not in memory, we just insert the index entry into a waiting list, and insert it into the respective index block next time the index block is retrieved into memory.

5.3. Other Indexes

To make an OODB efficient, we need other indexes in addition to the OID index, e.g., to implement collections. In Vagabond, these indexes are all realized as objects.

For temporal queries, it is useful to be able to search and retrieve by time as well as OID. A multidimensional index can be used here, e.g., time-split B-trees [9] or R-trees [4].

6. Physical Storage Structures

The log is stored on a *volume*. One volume is a configuration of one or more storage devices. The storage devices are typically disk partitions, cf. Figure 1. A volume consists of a volume information block, a number of equal sized segments, and a checkpoint region.

The size of the segments is set when the volume is formatted. 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, since we have to wait for transactions during group commit, and result in a larger number of subsegments (one or more subsegments can be written to a segment). Also, segment writing blocks read operations.

6.1. Volume Information Block

This block holds static volume information, and is only read when the system is started. It is written when the volume is formatted, and when new devices are added to the volume. Devices can be added to the volume while the system is running. Each device has its own volume information block. Static volume information includes segments size, number of segments, large object threshold, and locations of segments status blocks.

Pointer to next subsegment
Subsegment checksum
Number of ODs
Number of small objects
Number of large object index blocks
Number of large object subobjects
The number of OIDX index blocks
Number of transaction prepare start
Number of transaction commit finished
Number of transaction abort
Size of segment status table
[OIDX block]
[Large object index block]
[Large object subobject]
[OID + Object length (one tuple for each object)]
[OIDX block ID (one for each index block)]
[Large object OID + index block ID]
[Transaction control information]
[OD]
Segment status table

Figure 5. Segment structure. A field name written as *[FIELD]* is short for zero or more instances of *FIELD*.

6.2. Segment Structure

Objects, index blocks, transaction control information, and object descriptors, are placed into the segments. We do not always have a full segment of data to write when a flush is requested (timeout on group commit), and therefore we write data as subsegments.

The layout of a segment is shown on Figure 5. The first part is the segment header, consisting of some identifying information, link to next subsegment, and a subsegment checksum.

The main part of the segment consists of objects, index blocks 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, to be able to check if an object is still valid. If we did not store this information in the segment, we would have to search through the whole OIDX to find out which objects resides in the segment. This is obviously too costly. The solution is to store the OID and object length together with the object. Ideally we would store the OD here, but this is not possible. The reason is that objects can be written before a transaction commits because the buffer is full, or simply to avoid a heavy burst at commit time. If this is done, the transaction is not yet given its timestamp, and we do not have enough information to write the ODs. Therefore, they are not written until transaction commit. This, implies that in general, the object and its OD are in different segments.

The reason for writing both ODs and index blocks, is to avoid having to flush all index blocks at commit time. By writing these ODs, the index blocks themselves can be written lazily to disk later. If the system crashes, dirty index blocks that had not been written can be reconstructed from the information in the ODs at recovery time.

Transaction control information is written to the segment just as it would be done in an ordinary write ahead log, but no information is written until the transaction start the commit process.

Finally, at the end of the segment, we can have a segment status table. This table contains the status (clean, dirty etc.) for each segment, together with some access statistics to help decide victims in the cleaning process. This is kept in main memory during normal operation, but periodically written to the log as a part of the checkpointing.

6.3. Checkpoint Blocks

These blocks holds the location of the last written version of the segment status information, and pointers to the object index root. This is the starting point for the analysis pass when doing recovery.

7. Object Operations

In this section, we describe how objects in the log are accessed, and how transaction management and recovery is done.

7.1. Writing Objects

When we create a new object, it is allocated a unique OID. The buffer system employs a steal strategy, which means that objects can be written to disk before the transaction commits, if the buffer gets out of space. To ensure durability, the object have to be written to disk before the transaction commits. If it is an update (new version of an existing object), only a delta object needs to be written. In our system, we will write delta objects if some given criteria is satisfied, e.g., based on the difference between the size of the object and the delta object. In this way, we are guaranteed that delta objects will only be written if they are beneficial. This is important, since writing delta objects has the unfortunate disadvantage that after a crash, to read an object, we will have to go through a list of delta records to reconstruct the object. However, during normal operation, transactions will usually be able to commit much faster, and for objects that are hot spots, delta objects can save a lot of bandwidth.

The object's OD is not written until commit time (cf. Section 6.2).

7.2. Deleting Objects

Deleting an object is done by writing a tombstone version, which is an OD where the physical location is NULL, and the timestamp is the delete time.

If we do not want to keep the deleted version (it is not a temporal object), the object is simply marked as deleted in the index. The object will sooner or later be removed by the cleaning process.

7.3. Reading Objects

To read an object that is not resident in memory, we first have to look up in the OIDX to find the physical location, and then read the object. We expect that at least the upper levels of the index is resident in the buffer, so that most lookups will be satisfied from the buffer system, and that reading index blocks is only needed for a few of the lookups.

7.4. Transaction Management

When a transaction commits, it is necessary to write enough information in the log to be able to do recovery in case of a crash. Objects can be written to disk before a transaction commits, e.g., due to insufficient buffer space. However, it does not become visible before commit, which is done by updating the indexes. To avoid having to force the index blocks to disk, only object descriptors need to be stored before a transaction commits it finished. Commit of a transaction is done by writing the index entries, the ODs. The index blocks themselves, which consists of many (possibly unrelated) entries on each, can be written later, since they can be reconstructed at recovery time from the index entries written at commit time.

The easiest way to do implement commit in our system, would be to do commits serially (not to be confused with executing transactions serially), e.g., write all ODs from one transaction before writing ODs from the next, and write transaction finished marks between each of them. At recovery time, we would immediately know which transaction finished last. Since many transactions can be committed in one segment write, this would not give problems with throughput, but could give higher response times for some transactions, if a transaction with many created or modified objects was earlier in the queue. If done serially, all ODs of this transaction must be written first. Reordering could alleviate the problem, by writing the data from smaller transactions first. But sooner or later, we have to write the large one, and this would effectively block the system for a while. In most applications, we would be able to live with these problems, but there is one additional, more serious, problem: it would be difficult to implement an efficient 2-phase commit, since the system would be blocked during the whole process, from prepare to commit finished.

Our solution, is interleaved commit. The timestamp in the ODs act as a transaction identifier. When a transaction finishes the commit, this is noted in the log. Thus, at recovery time, it is easy to decide which transactions finished before the crash. With our scheme, two-phase commit has the same cost as a local commit, and actually, in our system, we only operate with two-phase commit. A local commit is a two-phase prepare followed immediately by a two-phase commit.

The fact that no transaction control information is written to the log before a transaction starts the commit process, simplifies abort considerably. This is especially important in a client-server environment. It is also useful to exploit optimistic concurrency control techniques.

7.5. Checkpointing

Efficient checkpointing is important. Currently, we have only sharp checkpointing in our design, but other strategies appropriate for our system will be considered. In general, sharp checkpointing is not a good alternative, since the system has to stop the normal processing while doing the checkpoint operations. However, in our system, *most operations can run as normal during checkpointing*. The only restriction is that ongoing commit operations have to finish before the checkpointing starts, and new commit operations have to be delayed until the checkpointing finishes. Unlike conventional systems, the checkpointing operations does not involve lots of random writes. Data and index blocks are written to the log as usual. The only non-sequential write is the update of the checkpoint block.

7.6. Segment Cleaning

As times go by, and non-versioned objects gets deleted, more and more of the space in the segments becomes garbage. The disk will eventually fill up, and we do not have any clean segments left. Before this happens, we move non-deleted objects in almost empty dirty segments, to the current segment. In this process, which is called cleaning, we get empty (clean) segments as a result.

There are some tradeoffs involved in the choice of *which segment to clean*. Several constraints have to be satisfied, e.g., free as much space as possible. It is also beneficial to cluster together objects that are expected to have the same lifetime, to avoid having to move the objects many times, and cluster together related objects. There is ongoing work in this area in the context of LFS [10], and some of the results

should be applicable here as well. However, the clustering constraints give us some additional problems, which is not as important in a file system.

7.7. Tertiary Storage Migration

Even if disk space is cheap, it will still be necessary to have data on tertiary storage for some applications. This can be done transparently in our system, as a part of the cleaning process. Index blocks, as well as objects, can migrate. It is possible to determine from the physical location addresses where objects and index blocks are stored.

7.8. Recovery

Crash recovery can be done fast and efficient. At system startup time, it is determined from the checkpoint block whether the the shutdown of the system was controlled or caused by a crash. If done controlled, resident structures are built from the data written during shutdown, the most important being the last segment status table and the index root blocks.

If crash recovery is needed, we start reading from the last checkpoint, and build the relevant structures in memory in one forward analysis pass. If we need to write objects or index blocks during recovery because of insufficient buffer capacity, this is done to a clean segment. Since we do not do any in-place update, idempotence is no problem. When we read a subsegment that was only partially written, we have come to the place where the system crashed. We do a checkpoint, and when the checkpoint process is finished, the checkpoint blocks are updated. Because we do not modify any written data before updating the checkpoint block, idempotence is guaranteed. If the system crashes before the checkpointing is finished, the recovery will start from the same point next time we try.

Media failure in a log-only system, can be handled by the use of mirroring (RAID 1), RAID with parity blocks, or logging to another node.

8. Related Work

No-overwrite strategies have been used in shadow-paging recovery strategies earlier, e.g., in System R [1, 3], but with the limited buffer size at that time, the performance was not satisfactory. Postgres [15] also employed a no-overwrite strategy, but had also its performance problems, for several reasons, the most important being the buffer force strategy used.

Log structured file systems was introduced by Rosenblum and Ousterhout [11], an idea which has been further developed through the BSD-LFS [12] and Spiralog [6, 17] file systems. LFS has also been used as a basis for a high performance RAID [13], and for tertiary storage management [7]. LFS has been used as the basis for two other object managers: the Texas persistent store [14], and as a part of the Grasshopper operating system[5]. Both object stores are page based. To our knowledge, there have been no publications on other object based log-only OODB, based on LFS principles.

9. Conclusions and Future Work

The size of main memory in a typical database server is increasing at a high rate, and it is now possible to keep much of the data and index structures in main memory. The result is that most read requests and index searches can be satisfied from the buffer system. However, before a transaction can commit, modified data has to be written to disk, so that recovery can be done if the system crashes. With current read optimized database systems, this will become a bottleneck. To solve this problem, we need write optimized database systems.

In this paper, we have described the data structures and algorithms needed to realize a write optimized log-only OODB, in the context of Vagabond, an OODB currently being developed at our department. Vagabond is a transaction time temporal OODB, where focus is to provide database system support for applications which have earlier used file systems because of the limited data bandwidth in current database systems, typical examples are super computing applications and geographical information systems.

The design of Vagabond is now finished, and we are currently in the implementation phase. In addition to implementing the storage manager as described in this paper, we will pursue issues related to super computing database applications (especially storage of matrixes), geographical information systems (which can exploit our support for temporal/versioned objects), and efficient buffer structures in the context of the object buffer in Vagabond.

References

[1] M. M. Astrahan et.al. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2), 1976.

[2] 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, Minneapolis, MN*, 1994.

[3] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2), 1981.

[4] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *ACM SIGMOD*, June 1984.

[5] D. Hulse and A. Dearle. A Log-Structured Persistent Store. In *Proceedings of the 19th Australasian Computer Science Conference*, 1996.

[6] J. E. Johnson and W. A. Laing. Overview of the Spiralog File System. *Digital Technical Journal*, 8(2), 1996.

[7] J. T. Kohl and C. S. M. Stonebraker. HighLight: Using a Log-structured File System for Tertiary Storage Management. In *Proceedings of the USENIX Winter 1993 Conference*, 1993.

[8] 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.

[9] D. Lomet and B. Salzberg. Access Methods for Multiversion Data. In *Proceedings of the 1989 ACM SIGMOD*, 1989.

[10] J. N. Matthews, 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 (to appear)*, 1997.

[11] 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.

[12] 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.

[13] K. W. Shirriff. *Sawmill: A Logging File System for High-Performance RAID Disk Array*. PhD thesis, University of California at Berkeley, 1995.

[14] 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.

[15] M. Stonebraker. The Design of the POSTGRES Storage System. In *Proceedings of the 13th Conference on Very Large Databases*, 1987.

[16] M. Stonebraker. *Readings in Database Systems (2nd edition)*. Morgan Kaufmann, 1994.

[17] 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.