The Vagabond Approach to Logging and Recovery in Transaction-Time Temporal Object Database Systems

Kjetil Nørvåg
Department of Computer and Information Science
Norwegian University of Science and Technology
7491 Trondheim, Norway
email: Kjetil.Norvag@idi.ntnu.no

Abstract

In most current database systems, data is updated in-place. In order 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. Even if this is done as a batch operation, it can result in many non-sequential writes. In order to avoid this, another approach is to eliminate the database completely and use a log-only *approach. In this case, the log is written contiguously to the disk, in a no-overwrite way using large blocks. When using the log-only approach keeping previous versions comes almost for free, and this approach is therefore particularly interesting for transaction-time object database systems. Although the log-only approach in its basic form is relatively straightforward, it is not trivial to support features such as steal/no-force buffer management, fuzzy checkpointing, and fast commit. In this paper, we describe in detail algorithms and strategies for object and log management that make support for these features possible.*

Keywords: Object database systems, temporal database systems, logging, recovery

1 Introduction

Many emerging application areas for database systems demand efficient support for features that are difficult to support with today's systems. Common to many of these, is the need for management of complex objects, support for temporal objects, and support for querying changes (for example, updates since a particular time T). Examples of such application areas are XML/Web databases and geographical information systems. In the case of XML/Web databases, users might want to see changes in a collection of documents (or Web pages) since the last time they checked. In the case of geographical information systems, users frequently want to see an area as it was at a certain point in time, or what features a map included at a certain time.

In this paper we describe the Vagabond approach for efficient storage and management of temporal objects, which is based on a *log-only* approach. Using a log-only approach, updated data is written contiguously to the disk, in a no-overwrite way using large blocks. This is done by writing many objects and index entries, possibly from many transactions, in one write operation. This gives good write performance, but at the expense of read performance. However, with today's large main-memory buffers this is less of a problem than it was previously. In the log-only approach, the log is the final repository for the data. This differs from most current object database systems (ODBs), where data is updated in-place, i.e., updated pages are written back to the same location. In order to support recovery and increase performance, write-ahead logging can be used. This logging defers the in-place updates, but sooner or later the updates have to be applied, and this often results in the writing of a large number of pages.

Figure 1 summarizes the most important differences

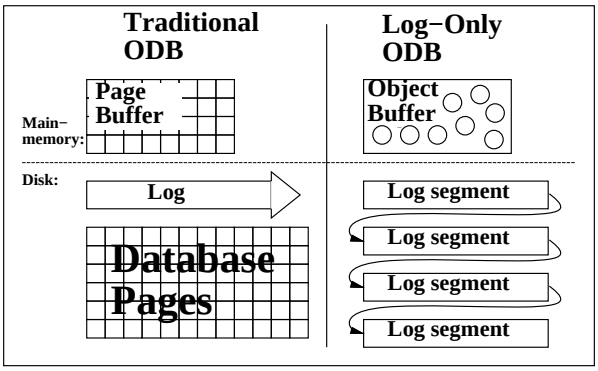

Figure 1. In-place update page server vs. log-only object database system.

between a traditional ODB and an (object-based) log-only ODB. In the traditional approach, pages containing objects are buffered in main memory, and recovery information is written sequentially to the log. The pages themselves are updated using random accesses to the disk. Even if this is done as a batch operation, it can result in many non-sequential writes. Using an object-based log-only ODB, an object buffer is naturally used, which gives a high memory utilization. Data is written sequentially to the log, giving a low write cost.

The log-only approach has many features that make it interesting. Two examples are fast recovery and ability to benefit more from using RAID technology than traditional systems (because of the writing of parity blocks, it is desirable that blocks to be written are much larger than those used in non-RAID systems). A third feature, which is the motivation for our research, is the fact that keeping previous versions of objects is a feature that comes almost for free when using a log-only approach. This is particularly interesting for transaction-time object database systems, where object updates do not make previous versions inaccessible. On the contrary: previous versions of objects can still be accessed and queried, and a system maintained timestamp (commit time of the transaction that created this version of the object) is associated with every object version. In a traditional system with in-place updating, keeping old versions of objects usually means that the previous version has to be copied to a new place before update. This doubles the write cost. When using a log-only approach, this is not necessary.

Using the log-only approach also gives new opportunities to improve performance. In order to reduce storage space and disk bandwidth, objects can be compressed before they are written. With the log-only approach, objects are written to a new location every time, so that *it is only necessary to use as much disk space as the size of the current version that is being written.* In a system that employs in-place updating it is difficult to benefit from object compression, because the compression ratio will differ from version to version, and it is difficult to know in advance how much space to reserve.

Previous log-only object database systems have been page-server based. While this works well in many contexts, it is not ideal. By operating on page granularity you get many of the disadvantages of traditional page servers. For example, if clustering is bad, and only a small part of a page has been updated, it is still necessary to write back the whole page. When this is the case, main-memory buffer utilization will be bad as well. A page-based log-only ODB also makes transaction management difficult. In order to avoid page level locking (note that from the concurrency-control point of view it is possible to use object-level locking, the problem is log management and recovery), you essentially need to 1) still have a separate log, possibly integrated into the segmented log, 2) use shadow paging, or 3) use ad-hoc techniques to solve the problem. All these solutions are likely to hurt performance and increase complexity. This has together with performance analysis of log-only temporal ODBs [13] convinced us that an *object based* log-only temporal ODB should be the preferred alternative.

Although the log-only approach in its basic form (using page granularity) is relatively straightforward, more effort is needed in order to achieve performance and high concurrency. In this paper, we describe the Vagabond approach for efficient storage and management of temporal objects. With the Vagabond approach, some of the problems in related designs are avoided, and steal/no-force buffer management, fuzzy checkpointing and fast commit are supported.

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 the Vagabond log-only approach. In Section 4 we describe in detail the algo-

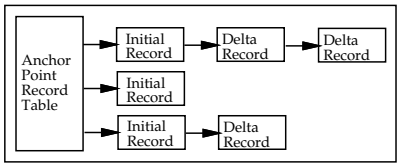

Figure 2. POSTGRES page.

rithms for the most important operations in the Vagabond approach. Finally, in Section 5, we conclude the paper.

2 Related work

No-overwrite strategies have been used previously in several approaches. One example is variants of shadow-paging recovery strategies that were used in early database systems, e.g., in System R [3]. However, the performance has not been satisfactory using these strategies. The main reasons for this, were limited buffer size in combination with unclustered data, and relatively large amounts of metadata that had to be written during the commit process.

POSTGRES [21, 22] also employed a no-overwrite strategy. Data in POSTGRES was stored in relations, which themselves were stored in files. Pages were allocated or deallocated for a file on demand, and were linked together. As illustrated in Figure 2, 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 updated, they were not updated in-place in the page. Instead, a delta record was created, which recorded the changes from the previous version. 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 (although a large object interface was added later), and delta records should be on the same page as the initial record. Although POSTGRES introduced many novel ideas, the performance was lower than expected. 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 pages modified by the transaction had to be written, giving a very high commit cost. In addition, a separate transaction log storing the state of all transactions had to be updated. It should be mentioned that the reason for this problem, was the assumption during the design of POSTGRES that stable main memory (non-volatile RAM) would be available. With stable main memory, the high commit cost could have been avoided.
- 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, thus increasing the overhead.
- In common with other no-overwrite strategies, POSTGRES also held the risk of declustering relations.

Other aspects that were not covered by the original design were integrated tuple/object identifier indexing (although this could be provided by an external index) and long transactions.

The no-overwrite idea was borrowed from POSTGRES and used in *log-structured file systems*, which the Vagabond approach is based on. Log-structured file systems were first introduced by Rosenblum and Ousterhout [16], and later refined by Seltzer et al. in BSD-LFS [18]. Log-structured file systems have also been the basis for other systems, for example Spiralog [24]. An interesting aspect of Spiralog is the B-tree integrated into the approach. However, although it could be used as a basis for indexing objects in an object manager, it was designed for file system updates, which are short transactions, and is less suitable for long-living transactions.

Log-structured file systems have also been used as the basis for several object managers: the Texas persistent store [19], as a part of the Grasshopper operating system [4], and the Lumberjack object store [5]. The storage managers in Texas and Grasshopper are page based, i.e., when an object has been modified, the whole page it resides on has to be written back. Similar to these approaches was Seltzer's implementation of transactional support in a log-structured file sys-

tem [17]. In her system, locking and updates were performed at page granularity, and all dirty buffers were held in memory until commit. Seltzer's study also included a performance comparison between a conventional system and a log-structured system. The results showed that the log-structured approach offered a performance improvement compared to the traditional approach.

To our knowledge, the only log-only ODB that is able to operate on *object* granularity (similar to the approach described in this paper) is Lumberjack. In Lumberjack, objects are stored in a *logical* log, managed by the client. The logical log is stored in paged segments. It is unclear how problems related to scalability and aspects of long-running transactions can be solved in that approach.

In the area of ODBs with integrated support for temporal objects, we are only aware of one prototype: POST/C++ [23]. POST/C++ is based on the partitioned storage approach, where current-version objects are stored clustered together, similar to a non-temporal ODB, and historical versions are stored separately from the current version. When an object is updated, the previous current version is copied to the *history store* before the new current version is updated in-place in the page. In addition, temporal ODBs built on top of non-temporal ODBs exist. One example is TOM, built on top of O_2 [20].

Storing data in a sequentially written log has been proposed as a way to store the historical versions in temporal databases [7]. The *log-structured history data access method* [9] uses some of the same ideas as in log-structured file systems and object stores. The log-structured history access method is based on the log-structured merge-tree, which is a hierarchy of indexes. Inserts and updates are only applied to the first level index, and the contents of one level in the index is 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 data, which is probably the most frequent operation also in a temporal database system, can be costly.

More details about the Vagabond approach can be found in the author's doctoral thesis [15].

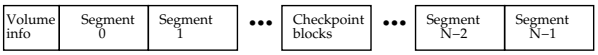

Figure 3. Disk volume structure.

3 The Vagabond approach

In this section we give an overview of our approach to log-only temporal ODBs. We give an overview of log management, storage objects, indexing, read- and write-efficiency related issues, and object access and queries.

3.1 Introduction to log-only log management

In the log-only approach, data that is already written is never modified. Instead, new versions of the objects are appended to the log. Logically, the log is an infinite-length resource, but the physical disk size is, of course, not infinite. As illustrated in Figure 3, this problem is solved by dividing the disk into large equal-sized physical segments. When one segment is full, writing is continued in the next available segment. As data is vacuumed, deleted or migrated to tertiary storage, old segments can be reused. Dead data, which in a temporal log-only ODB most frequently is old versions of index nodes, will leave behind partially filled segments. The data in these segments can be collected and moved to a new segment. This process, which is called *cleaning*, makes the old segments available for reuse. By combining cleaning with reclustering, it is possible to get well-clustered segments.

A segment can be in one of three states. 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, writing continues into a new segment. The new segment now goes from the *clean* state, to the *current* state. 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, which is the context where the term is more frequently used). Information about the state of the segments is kept in a *segment status table*. The segment status table is kept in main memory during normal operation, but is regularly checkpointed to disk, interleaved in the log, so that it can be recovered after a crash.

For each segment a *live-byte count* keeps track of how much of that segment that still is valid data. This live-byte count is decremented when data in a segment becomes invalid, for example when a non-temporal object is deleted. However, note that this is not the case when a temporal object is deleted. The previous version should still be available, so in this case the previous version is considered to be alive, and the live-byte count is not decremented.

At regular times a checkpoint operation is performed. In the checkpoint operation, enough information is written to the log in order to make it possible to use the current position in the log as a consistent starting point for recovery. The checkpoint information is stored in *checkpoint blocks*, which are stored in fixed positions in the log. Recovery in a log-only database can be performed very fast, since there is no need to redo or undo any data: at recovery time it is only necessary to do an analysis pass from the last known checkpoint to the end of the log where the crash occurred (or from the penultimate checkpoint in the case of our approach, where recovery time is slightly increased in order to increase performance and concurrency during normal operation).

3.2 Objects

In an object database, an object is uniquely identified by an object identifier (OID) (which can be as simple as an integer). When an object is updated, the new version has the same OID as the previous version. This is also the case in a temporal database. In order to distinguish between different versions of an object that have the same OID, the timestamps are used. In a transaction-time temporal database, the timestamp is the commit timestamp of the transaction that created or updated the actual object version.

Every object version has an associated object descriptor (OD), which contains the OID, physical location of the object in the log, commit timestamp (when the OD is not in the OIDX, the end timestamp of an object version is also included in the OD in order to reduce the cost of certain operations), and other administrative information. These ODs are used in the mapping from OID (and/or timestamp) to physical location on secondary storage, and stored in an *OID index* (OIDX).

In a non-temporal ODB with in-place updating of objects, the OIDX needs only to be updated when objects are created, not when they are updated. In a log-only ODB, however, the OIDX needs to be updated on every object update. This might seem bad, and can indeed make it difficult to realize an efficient non-temporal ODB based on this technique. However, in the case of a *temporal* ODB, the OIDX needs to be updated on every object update also if using in-place updating, because either 1) the previous or 2) the new object version must be written to a new place. Thus, when supporting temporal data management, the indexing cost is the same in these two approaches.

In our approach, all objects smaller than a certain threshold should be written as one contiguous object. Objects larger than this threshold are segmented into *subobjects*, and a *subobject index* is maintained for each of these large objects (this should be done transparently to the user/application). The subobject indexes used in the Vagabond approach are heavily based on the EXODUS large storage objects [2], which also take care of versioning of large objects. The value of the large object threshold can be set independently for different object classes. This is very useful, because different object classes can have different object retrieval characteristics.

The entries in the leaf nodes of the subobject indexes are *subobject descriptors*. The subobject descriptors are also stored together with the subobject in the segments. The contents of a subobject descriptor includes OID, physical location, and write timestamp.

3.3 OID indexing in a temporal ODB

In a traditional ODB, the OIDX is usually realized as a hash file or a B^+-tree, with ODs as index entries, and using the OID as the key. In a temporal ODB, there are more than one version of some of the objects, and it is necessary 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. The reason for having one index, instead of separate indexes for current and historical versions, is to reduce the update costs. Suitable indexes are, e.g., a traditional multiversion access method such as the TSB-tree [6], or the Vagabond temporal OID index, which is optimized for OID indexing

in temporal ODBs [14].

The most recently used OIDX pages are kept in a buffer pool in order to make OID mapping efficient. However, in an OIDX, index entries have in general low locality, i.e., only one or a few ODs in an index page are accessed before the page again is evicted from the buffer. In order to better utilize the memory, it is possible to keep the most recently used ODs in an *OD cache*.

In a database with many objects most of the ODs that are updated during one checkpoint interval[1] will reside in different leaf nodes in the OIDX. As a result, *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 accesses to the nodes will be random disk accesses, and as a result the installation reads are very costly. In order to improve update performance, the *persistent cache* (PCache) can be used. The PCache is an intermediate index structure, that contains a subset of the entries in the OIDX. The goal is to have the most frequently used ODs in the PCache. In addition, recently updated/created ODs are also stored in the PCache. In contrast to the OD cache in main memory, the PCache is persistent (i.e., its contents is not lost after a crash), so that it is not necessary 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 reduced OD update costs.

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 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. The entries in the PCache are maintained according to an LRU-like mechanism. The result should be higher locality on accesses to the PCache nodes, which reduced the total number of installation reads. The average OIDX lookup cost will therefore also be less than without using a PCache. Management information for the PCache, including the LRU access tables, PCache pointers, and OID ranges for all PCache nodes, are kept in a *PCache status table*. This table is regularly checkpointed to disk. A more detailed description and performance analysis of the PCache can be found in [12].

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

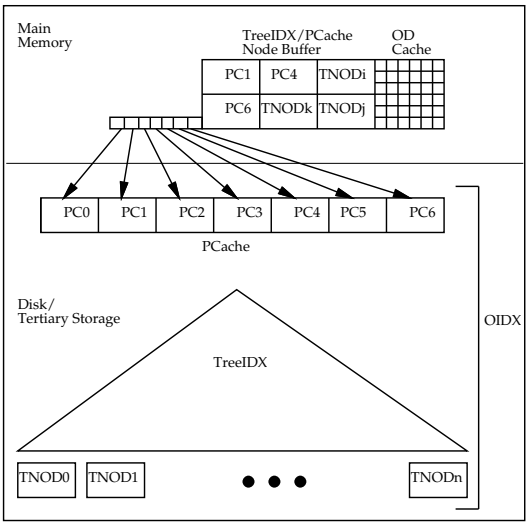

Figure 4. Overview of the TreeIDX, PCache, and index-related main-memory buffers. PCache nodes PC1, PC4 and PC6, and three TreeIDX nodes (denoted TNODn) are in the main-memory buffer.

In order to avoid confusion, we will in the rest of this paper denote the OID index tree itself as the *TreeIDX*, and use *OIDX* to denote the combined index system, i.e., the PCache and the TreeIDX. Thus, when we say an entry is in the OIDX, it can be in the PCache, in the TreeIDX, or in both. This is illustrated in Figure 4.

3.4 Read and write efficiency issues

A log-only ODB is write-optimized, and as a result, object retrieval is a potential bottleneck. Several techniques can be used in order to improve the read performance: 1) careful layout of objects, 2) hash-based signatures, and 3) object compression. 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, writing of delta objects can be employed to further reduce the amount of data that needs to be written to disk. This technique reduces the write cost, but might increase the read cost.

Careful layout of objects. Several strategies can be 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 using 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. This problem is similar to the general problem of reclustering of objects, and techniques from existing work is applicable here [8].

Hash-based signatures. A hash-based signature, generated by applying a hash function on the contents of some or all of the attributes of an object, can be stored in the OD. In a perfect match query, it is possible to determine from this signature whether an object is a possible match: a hash-based signature generated from the attributes in the query is compared with the signatures of the objects that are queried, and only objects with the same signature as the query signature can be possible matches. This comparison can be performed even before the objects are retrieved, so that in many cases it is possible to avoid the retrieval of the object themselves. The OD is accessed on every object access in any case, so that the additional signature-retrieval cost is only marginal. In addition, in a log-only ODB, the OIDX is updated every time an object is modified, so that additional signature maintenance cost is also small. As shown in [11], substantial gain can be achieved by the signature-in-OD approach.

Object compression. Objects can be compressed before they are written to disk. In this way, storage space and use of disk bandwidth can be reduced. With a log-only approach, objects are written to a new location every time. As a result, only as much space as the size of the version that is currently written is used. This contrasts to in-place updating approaches, where more space has to be reserved in case subsequent versions have a lower compression ratio.

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 between the versions are written. This is especially the case if an object is a hot-spot object, but it is also interesting as a way to reduce the storage requirements of a temporal database. 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.

3.5 Object access and queries

In non-temporal ODBs, ODMG's OQL or a similar query language 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, a temporal SQL-like query language. However, one of the main advantages of ODBs is the avoidance of the language mismatch by providing computationally complete data manipulation languages with no mismatch between language and storage. For ODBs, language bindings based on several languages exists, including C++ and Java. Such language bindings are also needed for temporal ODBs.

A general purpose programming language is only designed for current data. Integrating support for access to historical data into a programming language introduces a lot of interesting but difficult issues, including which object interface to use when accessing an historical object version (the schema might have been changed since the historical version was created), which method implementation to use when calling methods in historical objects, and how to integrate *time* into the syntax of the programming language. A more detailed discussion of these issues is given in [15].

4 Log-only database operations

In this section we present the algorithms for the most important operations in a log-only temporal ODB based on the Vagabond approach. We give an overview and an introductory example of log writing, and continue with more detailed descriptions of the different operations in the rest of the section.[2] First, we start with a summary of what should be stored in the log

[2] The description in this section is based on using magnetic disk as secondary storage. However, except for the device that stores the checkpoint blocks (see Section 3.1), other storage technologies can also be used.

segments and what buffers should be available in order to efficiently support the algorithms and strategies described in this paper.

The segments in the log should contain the physical address of the previous and next segment, object related information (including TreeIDX nodes, PCache nodes, ODs, small objects, subobjects and subobject-index nodes of large objects), transaction control information (including commit operations, as will be described in more detail in Section 4.3.1), and parts of persistent copies of the main-memory tables (segment status, PCache status, and transaction identifier/-timestamp/counter tables).

The information to be written to a segment is in general collected from the relevant main-memory buffers. The most frequently used ODs and the ODs not yet installed into the OIDX should be stored in an OD cache, as illustrated in Figure 4. In order to reduce index access cost, the most frequently used PCache and TreeIDX nodes should also be buffered in main memory. In addition to the index-related buffers, modified or recently used small objects should be buffered in a small object buffer. As described in Section 3.2, large objects can be segmented into subobjects, and recently used or modified subobjects and subobject index nodes should be buffered in dedicated buffers. All buffers can be maintained according to a LRU policy.

4.1 Introduction

When a transaction[3] is started, it is assigned a transaction identifier (TxID). Unlike many other systems, it is not necessary to write information to the log when a transaction starts. 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.

Modified objects can be written to the log before a transaction commits (*buffer steal strategy*). This reduces commit time (the objects are already safe on disk when the commit operation starts), as well as making it possible to handle large amounts of data in one transaction (the amount of main memory does not limit the amount of data a transaction can create/update).

When an object is written to the log, it is always written together with its OD. This OD is only intended to be used if crash recovery is needed. The commit timestamp is obviously not known before commit time, so that when objects are written to the log the ODs in the log contain the TxID instead of the timestamp. When a transaction commits, a `(TxID,timestamp)` tuple for the committing transaction is written to the log as a part of the commit operation. In this way, it is possible to know the timestamp of a committed transaction when doing recovery.

After a transaction commits, the objects that have been created or updated by the committing transaction become current versions, and should be available for other transactions to read. When another transaction later wants to read an object, it has to first retrieve the OD of the object. This OD is either still in the OD cache (see Section 3.3), or has been installed into the OIDX. During normal operation, ODs from a committed transaction are lazily installed into the OIDX after the commit has been finished, and ODs are never discarded from main memory before they have been installed into the OIDX.[4] When the ODs are inserted into the OIDX, the timestamp is used in the OD, and not the TxID.

In order to bound the amount of log that has to be processed during crash recovery, all ODs generated from a transaction that commits during one checkpoint interval (between two consecutive checkpoints), should be installed into the OIDX before the next checkpoint interval ends (if this is not the case, the ODs will be written at that time, before the checkpoint operation can finish). This implies that during recovery, we know that we only have to process log back to the penultimate checkpoint (the actual length of a checkpoint interval can be application specific, and in general, it will be a tradeoff between performance and recovery time).

We denote a transaction as *short* if all the created objects and ODs from the transaction, *and* the commit operation, are written during one checkpoint interval. If a transaction lasts longer than one checkpoint inter-

[3] All transactions in this paper are assumed to be flat transactions having ACID properties (atomicity, consistency, isolation, and durability).

[4] Remember that we denote the index tree itself as the TreeIDX, and use OIDX to mean the combined index system, i.e., the PCache and the TreeIDX. Thus, when we say an entry is in the OIDX, it can be in the PCache, in the TreeIDX, or in both (see Section 3.3).

val, 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 still guaranteed to be installed into the OIDX during the 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 (this will be described in more detail in Section 4.2.1). Entries from aborted transactions will be lazily removed from the PCache nodes as the PCache nodes are retrieved from disk later. Objects in the log from aborted transactions will simply be discarded the next time the segments are cleaned.

An interesting point that should be observed is that nothing from a transaction is written to the log 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 before commit time, even if it has a long duration. This implies that in a system based on the principles described in this paper, many long transactions can be considered similar to short transactions (but note that problems related to concurrency control, e.g., lock contention, still remain).

Fast crash recovery is one of the advantages of a log-only system. If crash recovery is needed, the last part of the log is scanned. Because all ODs generated from a transaction that commits during one checkpoint interval should be installed into the OIDX when the next checkpoint interval ends, only the log written after the penultimate checkpoint has to be processed. ODs from committed transactions that are not yet installed into the OIDX are collected when the log is read, and can later be installed into the OIDX. ODs from aborted transactions and transactions that were ongoing at crash time, are discarded.

4.1.1 Example

We will now illustrate log writing with the use of Figure 5, which shows a number of transactions T_i. On the top, there is the time-line, running from left to right, with checkpoints marked. The ODs are written together with the objects, and installed into the OIDX at a later time (note that at the time ODs are installed into the OIDX, they are still in main memory), as illustrated in the figure. Please note that even though the transactions are illustrated with separate lines, objects and ODs from different transactions, as well as nodes of the OIDX, can be stored in the same segments.

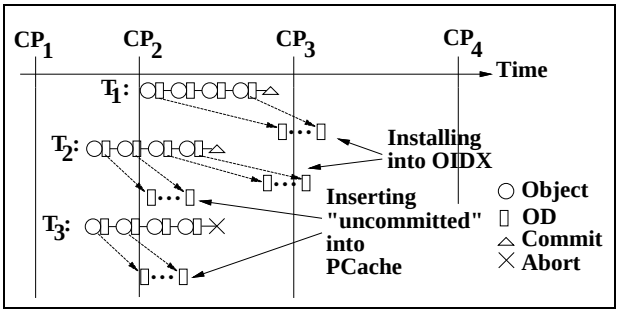

Figure 5. Example of log writing. In the figure, an object written to the log is illustrated with a ◯, an OD as a □, the commit operation by a △, and the abort operation with a ×.

Starting with transaction T_1, this is a short transaction. The transaction commits during the second checkpoint interval between checkpoint 2 and 3, and the ODs generated from this transaction should be installed into the OIDX when checkpoint 4 ends. In this way, it is guaranteed that in the case of a crash after checkpoint 4, no uninstalled ODs from transaction T_1 will exist in the part of the log written before checkpoint 3.

Transaction T_2 spans more than one checkpoint interval, and is therefore treated as a long transaction. All the ODs written by transaction T_2 during the first checkpoint interval (before checkpoint 2), must be installed into the PCache before the end of checkpoint 3. This might be postponed until the end of checkpoint period 2, but if the ODs have to be replaced from main memory, they can also be installed "uncommitted" into the PCache in the background during the second checkpoint interval. After the transaction has committed, its ODs can be inserted into the TreeIDX as well. To emphasize: Before transaction T_2 commits, its ODs can only be inserted into the PCache, but

after the transaction has committed, its ODs can be inserted into the TreeIDX as well as the PCache. This will be explained in more detail in Section 4.2.1.

Similar to the case of transaction T_2, some of the ODs from transaction T_3 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.

We have now given a short introduction to the log generation, and will continue with a more detailed description of the operations.

4.2 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, and 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 TreeIDX before the actual transaction commits, but in the case of long 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.

4.2.1 OD management

In traditional applications, most transactions are *short* and update only a few objects. In the approach described in this paper, the ODs that the short transaction have generated are in general not discarded from main memory until they have been installed into the OIDX,

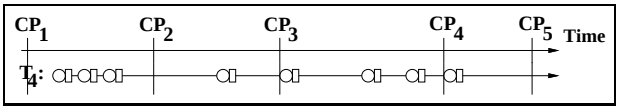

Figure 6. Long transaction.

and the ODs written together with the objects in the log will only be used later if crash recovery have to be performed. In addition, all ODs resulting from a transaction committed during one checkpoint interval, should be installed into the OIDX before the next checkpoint interval ends. However, in the case of *long*[5] transactions, which can be common in applications using ODBs, the number of ODs can be very high. If all ODs are required 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 the 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. This is in many situations not acceptable.

The problems with long transaction problems are illustrated in Figure 6, which shows an example of a long transaction. If 1) ODs are not installed into the OIDX before commit time[6], 2) transaction T_4 commits during checkpoint interval 5, and 3) 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 that had not been installed into the OIDX when the crash occurred. Some possible solutions to the problems with long transactions are to 1) use a larger OD cache, 2) use one or more segments as "sidefiles" to store the ODs from long transactions, or 3) allow ODs from uncommitted transactions to be inserted into the PCache. We will now discuss these alternatives, and why we consider the third alternative as the best solution.

[5] When we in this section study long transactions, we mean both traditional long-living transactions, as well as "large transactions", i.e., transactions that generate large amounts of data.

[6] Instead of using the approach of storing uncommitted ODs as described previously, and in more detail below

Large OD cache. Solving the long transaction problem by increasing the size of the OD cache has several drawbacks:

- Increasing the OD cache means less memory available for buffering OIDX nodes and objects. This increases the buffer-miss rates, which in turn reduces 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 can 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 to be processed would be from transactions that have already been committed and have their ODs inserted into the OIDX, which means that most of the log processing is duplicating earlier work.

For these reasons, we do not consider a large OD cache as a reasonable solution to the problem.

Sidefile segments. One or more segments could be used as "sidefiles", which are segments that are used only for storing the ODs generated by one particular transaction. The sidefile segments of the transaction 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, locking a large number of ODs in the OD cache is avoided, and the amount of log that needs to be processed during recovery is reduced. 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 there is 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 as well as its OD might have been written back to disk and removed from the buffer due to buffer replacement. In this case, the next time the object is modified, it is difficult to know that it has already been modified by the same transaction. A result could be 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_i 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_k tries to access objects written by T_i, 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 with ODs 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 the transaction 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 the 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 con-

sider 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 least 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 also only marginal.

We do not allow ODs from uncommitted transactions to migrate further from the PCache to the TreeIDX. 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.

4.2.2 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, it must be possible 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 TxIDs instead of timestamps. The problem can be solved in several ways, for example by using one bit in the OD as a flag to tell whether it belongs to a committed transactions or not, or to use a separate bitmap in each PCache node in order to know whether an OD in a slot is from a committed or a uncommitted transaction. However, a better solution is to store the ODs in a PCache node 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 the other tree, the *uncommitted tree*, is used for ODs from uncommitted transactions. 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 that 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 OD is moved, the TxID in the OD is replaced with the commit timestamp of the transaction that generated the OD.

It is necessary to keep the TxID of a committed transaction until all ODs that were stored in the PCache before the transaction committed have been moved to committed trees. For each committed transaction, a counter of how many ODs from the transaction that are still in uncommitted trees in the PCache is maintained. Every time we move an OD from an uncommitted tree to a committed tree, this counter is decremented. When the counter reaches zero, information about this transaction can be discarded. The `(TxID,timestamp,counter)` tuples are stored in a TxID/timestamp/counter table. Entries from this table are written to the log during each checkpoint interval, in order to make it possible to reconstruct the table during recovery.

4.2.3 Creating and updating objects

When an object is created, 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 previously. 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 or not.

When large objects are updated, only the modified subobjects and the affected subobject-index nodes are written to the log. The use of the versioned subobject index (see Section 3.2) 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 achieved by first writing the updated subobjects, and then the modified parts of the subobject index.

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 and/or is compressed.

Delta objects. As described in Section 3.4, a delta object is the difference between the new and the previous committed version of an object, and by writing delta objects instead of the complete objects, the amount of data that has to be written to the log is reduced.

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 the current version 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. 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 certain criteria are satisfied, for example that the size of the delta object should be much smaller than the size of the complete object, and accesses to historical versions of the actual object should be infrequent.

Compressed objects. When a compressed object is written to the log, the uncompressed size of the object is included in the object. This makes it possible to know how much buffer space that has to be allocated before the object is decompressed. The size stored in the OD is the compressed size of the object, i.e., how much space the compressed version occupies on disk. This information is needed when the object is to be read from disk.

4.2.4 Deleting objects

Temporal objects are not physically deleted. In this respect, they are mostly treated as non-deleted objects. For example, during cleaning, a version of a deleted temporal object will be moved to the new segment, similar to a non-deleted object. A non-temporal object, on the other hand, will 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.

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 segment status table, see 3.1) 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 were intermediate delta objects, the live-byte counters for the segments where these delta objects reside are 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 the 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.

4.2.5 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, and in order to read an object that is not resident in memory the object's OD has to be retrieved first. 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 index has to be traversed in order to retrieve the requested subobject(s).

- **Current versions.** When reading the current version of an object based on OID, a lookup in the OD cache is performed to check if the OD is resident in the OD cache. If not, a lookup in the OIDX is necessary. An OIDX lookup is performed by first searching the PCache, and if the OD is not found in the PCache, the TreeIDX is searched. When doing a lookup in a PCache node, the uncommitted tree only has to be searched 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 more coarser granularity, for example a container/set of objects). When the OD is found, the object is read from the physical location found in the OD.

- **Historical versions.** If the timestamp of the historical version that is to be retrieved is known, the lookup for the OD and retrieval of the object can be performed 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, the OD with *the largest timestamp less than or equal to* t_j has to be retrieved. It is this operation that makes an *end timestamp* in the OD beneficial when the OD is outside the TreeIDX (see Section 3.2). If the end timestamp was no 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 was 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 it would be necessary to do a lookup in the TreeIDX for every such retrieval.

Delta objects and compressed objects. As described above, the OD has to be retrieved before an object can be retrieved in order to determine the physical location where the object is stored. The OD also indicates if the actual object version is a delta object or is compressed (except in the case of large objects, where this information is stored in the subobject de-

scriptors).

Delta objects. If we want to retrieve an object version V_j that is stored physically as a delta object, the actual version has to be recreated by retrieving the most recent complete version V_c that was created before V_j, in addition to retrieving the delta objects D_i that were written between V_c and V_j. This is achieved by first retrieving the ODs of the delta objects D_i and the OD of the complete version V_c. This is done by searching the OIDX backwards from version V_j to V_c (this operation will not be very costly, because the number of delta objects between two complete versions should be relatively small, and the ODs will be clustered in the OIDX). After the ODs have been retrieved, the actual object version V_c and the delta objects are retrieved, and the 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 that was written.

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.

4.3 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 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 ODs are written to the log in order to avoid synchronous updates of OIDX nodes at commit time. This section will give a more detailed description of transaction management in the Vagabond approach.

4.3.1 Commit

The transaction commit operation can in principle be implemented by first writing to the log the objects from the transaction that is still dirty in the object buffer, followed by a transaction finished mark which includes a `(TxID,timestamp)` tuple. After the objects and the transaction finished mark have 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. Although this technique in most cases should give good throughput in a single server system, it is possible that in the case of transactions that have generated a large number of ODs the result could be a queue of transactions that want to commit. In that case, smaller transactions could be blocked for a long period. However, the most important problem is that it would be *difficult to implement an efficient 2-phase commit operation by using this simple technique.* 2-phase commit is crucial to ensure consistency in multi-server systems where data is distributed, and to support it a more elaborate commit protocol has to be used, where more information is written to the log in the various phases of the commit process.

2-phase commit. When a transaction is started on node NodeID$_c$, it is allocated a transaction identifier TxID$_c$ on the node where it is started. If the transaction TxID$_c$ accesses other servers, a transaction TxID$_p$ is started on each of these servers. The transaction on NodeID$_c$ maintains a table of all subtransactions TxID$_p$ that it controls.

At commit time, the controller (TxID$_c$) sends a prepare message 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 considers the commit finished when all acknowledge messages have been received. In more details, the commit algorithms are as follows (also illustrated in Figure 7):

Controller:

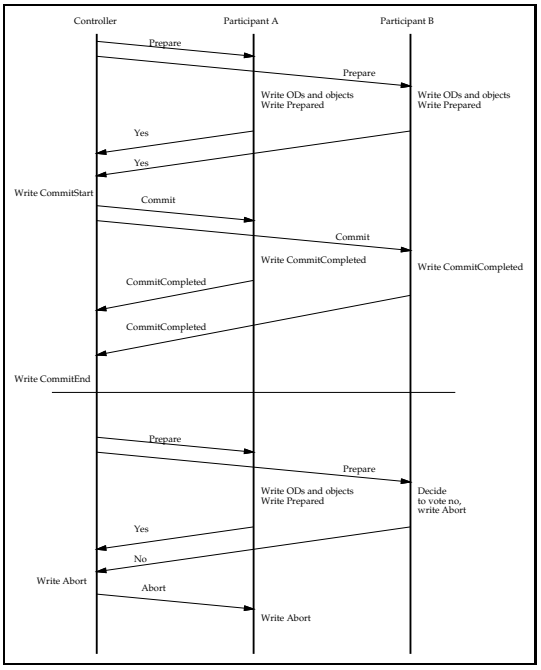

Figure 7. Messages and transaction information written to the log during a 2-phase commit involving a controller and two participants. On the top is illustrated a commit where all participants vote yes, and the commit succeeds. On the bottom is illustrated a commit where one of the participants votes no, with an abort as the result.

1. Send the `Prepare(TxID`$_p$`,TxID`$_c$`)` message to all participating nodes.

2. Wait for reply from all participating nodes.

3. (a) If all participating nodes vote yes, write `CommitStart(TxID`$_c$`,timestamp)` to the log, and send `Commit(TxID`$_p$`,timestamp)` messages to the participating nodes. When all participating nodes have acknowledged with `CommitCompleted`, a `CommitEnd(TxID`$_c$`)` 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(TxID`$_c$`)` is written to the log, and `Abort(TxID`$_p$`)` messages is sent to the participating nodes that voted yes.

Participating nodes:

1. (a) When the `Prepare` message is received and the participating node decides to vote no, `Abort(TxID`$_p$`)` 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(TxID`$_p$`,timestamp, NodeID`$_c$`,TxID`$_c$`)` is written to the log.

3. When the `Prepared` is safe on disk, a `Yes(TxID`$_p$`,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(TxID`$_p$`,`

`timestamp)` is written to the log, and a `CommitCompleted(TxID`$_p$`)` message is sent to the controller. Only after this has been done the transaction is considered committed, and created/modified ODs can be installed into the TreeIDX.

(b) If an abort message is received, transaction TxID$_p$ is aborted, and `Abort(TxID`$_c$`)` is written to the log. It is not strictly necessary to write an abort record to the log using our approach, but by doing so we avoid having to ask the controller node of the outcome after a crash.

If the transaction TxID$_c$ has not accessed data on other nodes, the commit can by done by simply writing `CommitCompleted(TxID`$_c$`,timestamp)` to the log after the dirty objects and ODs have been written, possibly in the same segment. Failure of one of the nodes during the 2-phase commit process can be handled in the same way as in traditional systems.

4.3.2 Abort

In a log-only database, it is not necessary to undo operations when a transaction is aborted. If the transaction that wrote an object does not commit, an object written to the log before the abort operation 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 TreeIDX until after the commit has completed. ODs with OIDs that have been allocated by a transaction that has aborted will never be inserted into the TreeIDX, 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 4.2.2).

When a transaction aborts, the live-byte counts in the segment status table for the segments where the objects were written are decremented accordingly. This can only be done immediately for the objects whose ODs are still in main memory. For those objects that 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 performed only if the validation phase succeeds.

4.3.3 Other transaction models

The description of transaction management has been presented in the context of flat ACID transactions. 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.

4.4 Recovery

When a system is restarted, it is determined from the checkpoint block 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 describe how to reduce the recovery time by checkpointing, and take a closer look at recovery and how to handle media failures.

4.4.1 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 database system, 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, the log is the final repository, and objects have to be written to the log before commit in any case. In this context, the main issue of checkpointing is to install the ODs into the OIDX. In this way, the amount of log that has to be read at recovery time in search of ODs that have not been installed into the OIDX before the system crashed, is reduced.

During a checkpoint interval (between two checkpoints), the ODs from committed transactions are installed into the OIDX. In order to keep the installation rate high enough and reduce the amount of memory

needed to store ODs that are 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, PCache status, and TxID/-timestamp/counter tables (see Section 3.1, 3.3, and 4.2.2) are resident in main memory. In order to be able to recreate these tables after a crash, the contents of these tables is written regularly to the log. Each time a segment is written, a certain range of entries from these tables are stored in the segment. During each checkpoint interval, all entries from these tables 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. Using the Vagabond approach, 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 that 1) were created before the last checkpoint and 2) are not yet installed into the OIDX, stop all other log processing until they have been installed. Note that this delay is undesirable, and can normally be avoided by giving high enough priority to OIDX updating. In order 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 the last checkpoint.

3. If there are entries in the segment status, PCache status, or TxID/timestamp/counter tables that have not been written during this checkpoint interval, write them to the log now.

4. Update the least recently written checkpoint block. This finishes the checkpointing, and by definition starts a new checkpoint interval.

4.4.2 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, undo or redo of objects is not necessary. However, the ODs and transaction management information in the tail of the log have 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 achieved by reading the log from the last checkpoint until one of the following conditions is satisfied: 1) the segment that is read was only partially written (the system crashed when it was writing this segment), or the *next segment* of a segment does not exist (every segment contains the address of the next segment, and if this segment does not exist, this means that the system crashed in the interval between writing two segments).

When the log is read in order to find the end of it, all ODs that are read are collected. Those ODs where we later find a commit record for the transaction that generated the ODs, are kept. ODs that do not have a corresponding commit record can safely be discarded because the system crashed before the actual transactions committed.

After the end of the log has been identified and the part of the log written after the last checkpoint has been processed, the log from the last checkpoint and backwards is read, until the penultimate checkpoint. In this way, all segments that might have ODs from committed transactions but where the ODs have not yet been installed into the OIDX, are processed. This backward reading can be done efficiently because all segments have a pointer to the previous segment.

While reading the segments, the relevant structures are rebuilt in memory. If it because of insufficient buffer capacity is necessary to write index nodes during recovery, these nodes are written to clean segments. When the log has been processed, a checkpoint is performed, and when the checkpoint process is finished, the checkpoint blocks are updated. Idempotence is guaranteed because no written data is modified before updating the checkpoint block. If a system crashes during recovery, it will simply start recovery

in the same way next time.

Media failure in a log-only system can be handled by the use of mirroring (RAID 1) or RAID with parity blocks (for example RAID 4 or RAID 5). The use of mirroring will also improve read performance, because the read bandwidth is doubled. The write performance will stay the same. Similar to traditional systems, disaster recovery can be supported by additional backup of logs, or logging to remote nodes.

4.5 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 previously been *logically* deleted (deleted temporal objects), and non-current (historical) versions of data. During vacuuming, all objects in a certain object class or in a certain container (set of objects), created before a certain time t_v, are removed. After vacuuming, these objects can not be accessed anymore, not even in historical queries. Vacuuming can be performed according to two different strategies, 1) *eager* or 2) *lazy* vacuuming.

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 because the ODs of all objects in a container are clustered in the OIDX, 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.

Lazy vacuuming. If vacuuming is done lazily, the physical removal of an object is deferred until the segment it resides in is cleaned. For each object class, a vacuuming age t_v can be defined, so that at cleaning time a non-current object will be discarded if it is older than t_v.

With lazy vacuuming, it is 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 of a segment will not be decremented before the segment is cleaned. If a segment consists of objects from the same class, created at the same time, the system would never discover that it could be cleaned because it would appear to be full with data that is still alive.

Choosing a vacuuming strategy. From the discussion above, it is clear that eager vacuuming should be used when possible, i.e., when it is known in which container(s) the objects to be vacuumed are stored. This is further justified by the fact that lazy vacuuming increases the complexity of the cleaning algorithms.

4.6 Segment cleaning

In a log-only database system, 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 updated and rewritten, the previous version becomes outdated. Without intervention, the disk volume will eventually fill up, and there would not be any clean segments left. As described in Section 3.1, this situation is avoided by cleaning segments. During cleaning, data[7] that is still valid is moved from segments that have mostly dead data, and written into new segments. The result of this process is empty (clean) segments.

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.

[7] Note that "data" in this context includes objects and subobjects, as well as index nodes and subobject-index nodes.

- Historical versions of temporal objects that have not yet been vacuumed (i.e., are younger than the vacuuming age for the actual object class in the case of lazy vacuuming, or the OD of the version is still in the OIDX in the case of eager vacuuming).
- Subobject and subobject-index nodes that are reachable from a large object that is still alive.
- OIDX nodes that 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 during cleaning.

Only dirty segments written before the penultimate checkpoint can be cleaned. The reason for this, is that parts of the segment status, PCache status, and TxID/timestamp/counter tables 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.

In the cleaning process, there are 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 (in order to improve read performance), and 3) cluster together data that is expected to have the same lifetime, in order to avoid cleaning 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 dynamic clustering in the context of traditional ODBs, and results from related research in that area is applicable. 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 also be difficult.

Due to space constraints we can not go into details about cleaning-conscious segment creation, but to give an idea of possible strategies we note that the expected lifetimes for PCache nodes, TreeIDX nodes and objects are different. By writing data of only one category into a particular segment when possible, the cleaning cost can be reduced. A more general discussion about cleaning techniques for log-structured storage systems can be found in [1, 10].

4.7 Query Processing

Queries are essentially lookup operations, either on single objects, or on a set of objects. In a temporal ODB, queries can also be performed on time validity as well as values. The query processing can be considered as a mapping from a query into a set of result objects. From the storage manager point of view, the job is to deliver a set of candidate objects selected through the query processing with the help of available indexes.

Although the techniques for query processing in a temporal database system are not fundamentally different from the techniques in non-temporal systems, there is one aspect that should be emphasized: in a non-temporal system many queries can be efficiently based on a scan through a selection of tuples or objects. In a temporal query this will in general not be the case, and an efficient temporal OIDX will be of a high importance (the use of such an index in queries is described in [14]). The details of query processing are outside the scope (and available space) of this paper, but a survey of related work can be found in [15].

5 Conclusions and further work

The log-only approach for temporal ODBs has been shown to be promising with respect to performance. However, although the log-only approach in its basic form is relatively straightforward, achieving performance and high concurrency is not straightforward. In this paper, we have described in detail how this can be achieved, resulting in a design that avoids some of the problems in previous related designs. The results include support for steal/no-force buffer management, fuzzy checkpointing, and fast commit. This is in particular made possible by novel use of the PCache structure.

Based on results from an analytical performance analysis of the Vagabond log-only approach [13], we consider this approach very promising for the future, and we expect the benefits of the log-only approach to increase with increasing amounts of main memory that will be available in future systems.

The results from this paper, together with our other work in this context, has shown that the log-only approach is feasible. Some ideas that we would like to

explore further are the study of a valid time or bitemporal ODB using the log-only approach, and parallel systems based on the log-only approach.

Acknowledgments

I would like to thank Kjell Bratbergsengen and Olav Sandstå for useful discussions and constructive comments.

References

[1] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceeding of the Winter 1995 Usenix Conference*, 1995.

[2] M. Carey et al. Object and file management in the EXODUS extensible database system. In *Proceedings of the 12th VLDB Conference*, 1986.

[3] J. Gray et al. The recovery manager of the System R database manager. *ACM Computing Surveys*, 13(2), 1981.

[4] D. Hulse and A. Dearle. A log-structured persistent store. In *Proceedings of the 19th Australasian Computer Science Conference*, 1996.

[5] 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.

[6] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD*, 1989.

[7] D. Lomet and B. Salzberg. Exploiting a history database for backup. In *Proceedings of the 19th VLDB Conference*, 1993.

[8] W. J. McIver, Jr. and R. King. Self-adaptive, online reclustering of complex object data. In *Proceedings of the 1994 ACM SIGMOD*, 1994.

[9] P. Muth, P. O'Neil, A. Pick, and G. Weikum. The LHAM log-structured history data access method. *VLDB Journal*, 8(3-4):199–221, 2000.

[10] J. M. Neefe et al. Improving the performance of log structured file systems with adaptive methods. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, 1997.

[11] K. Nørvåg. Efficient use of signatures in object-oriented database systems. In *Proceedings of ADBIS'99*, 1999.

[12] K. Nørvåg. The Persistent Cache: Improving OID indexing in temporal object-oriented database systems. In *Proceedings of the 25th VLDB Conference*, 1999.

[13] K. Nørvåg. A comparative study of log-only and in-place update based temporal object database systems. In *Proceedings of CIKM'2000*, 2000.

[14] K. Nørvåg. The Vagabond temporal OID index: An index structure for OID indexing in temporal object database systems. In *Proceedings of IDEAS'2000*, 2000.

[15] K. Nørvåg. *Vagabond: The Design and Analysis of a Temporal Object Database Management System*. PhD thesis, Norwegian University of Science and Technology, 2000.

[16] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1), 1992.

[17] M. Seltzer. Transaction support in a log-structured file system. In *Proceedings of the Ninth International Conference on Data Engineering*, 1990.

[18] M. Selzer et al. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Conference*, 1993.

[19] 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.

[20] A. Steiner. *A Generalisation Approach to Temporal Data Models and their Implementations*. PhD thesis, Swiss Federal Institute of Technology, 1998.

[21] M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 13th VLDB Conference*, 1987.

[22] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE TKDE*, 2(1), 1990.

[23] 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.

[24] 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.