Issues in Transaction-Time Temporal Object Database Systems

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

Abstract

Object database systems (ODBs) are an attractive alternative to relational database systems, especially in application areas where the modeling power or performance of relational database systems is insufficient. These applications typically maintain large amounts of data. Frequently, some of the data is temporal data. For the temporal data, the whole history of the individual objects is kept, and data is never deleted. The area of temporal ODBs is still immature, and there are many design issues that need to be solved in order to be able to achieve the desired performance. In this paper, we discuss some temporal ODB research issues and possible solutions related to object storage, object management, main memory buffering, and language bindings.

INTRODUCTION

Object database systems (ODBs) are an attractive alternative to relational database systems, especially in application areas where the modeling power or performance of relational database systems is insufficient. These applications typically maintain large amounts of data, and additionally, often want to manage temporal data. For the temporal data, the whole history of the individual objects is kept, and data is never deleted.

Application areas of temporal object database systems include GIS systems (geographical information systems), scientific and statistical databases, multimedia systems, PACS (picture archiving and communications systems), and XML warehouses (for example Xyleme (Aguilera et al., 2000)).

The area of temporal ODBs is still immature, and most of the work has been done on data models and query languages. However, designing and implementing a system is something completely different, and introduces new problems that have to be solved. In this paper, we discuss problems and possible solutions derived from the Vagabond project at the Norwegian University of Science and Technology. We will concentrate on issues not covered by previous work (discussed in the related works section), and in particular consider issues which are particular for temporal *object* database systems, compared to temporal *relational* database systems.

Transaction-Time Temporal Object Database Systems in a Nutshell

In a transaction-time temporal object database system (ODB), every object is associated with time, and an object can exist in several versions, each version being valid in a certain time interval. Every update creates a new object version. The new version is called the *current version*, while the previous versions are called *historical versions*. This versioning, related to time, is supported and maintained by the system. The system also provides support for querying the temporal data.

In a non-temporal ODB, space is allocated for an object when the object is created, and updates to the objects are done in-place. This implies that after an object update, the previous version of the object is not available. In an ODB, an object is uniquely identified by a logical object identifier (OID), and an OID index (OIDX) is used to map from the OID to the physical location of an object (Some systems use a physical OID, which means that the disk page of the object is given directly from the OID. However, in a temporal ODB, logical OIDs is the only reasonable alternative, because of objects being moved.) The entries in the OIDX, the *object descriptors* (ODs), contain administrative information, including information to do the mapping from logical OID to physical address. In a non-temporal ODB, the physical location of the new version is the same as the previous version, hence, the OIDX needs only to be updated when objects are created and when they are deleted. In a temporal ODB on the other hand, we have to either 1) write the new current version to a new location, or 2) copy the previous version to a new location before we update the current version in-place. In any case, we have to update the OIDX every time we update an object, and we use one OD for each object version.

Outline of Paper

The organization of the rest of the paper is as follows. First, we give an overview of related work. In the following section we discuss object management issues, including object storage alternatives, main memory management, OID indexing, and temporal clustering and access patterns. Then, we discuss issues related to object access and queries, including programming language bindings. Finally, we conclude the paper and outline issues for further research.

RELATED WORK

Even though temporal databases have a long history, few full scale systems have been implemented. Common for most of these, is that they have only been tested on small amounts of data, which make the scalability of the systems questionable. In most of the application areas where temporal database systems are needed, scalability is an important issue, as the amount of data will be large. In the area of temporal object database systems, we are only aware of one prototype, the POST/C++ temporal object store (Suzuki and Kitagawa, 1996), based on the Texas persistent store (see below). There have also been prototypes implemented on top of non-temporal ODBs, for example by Steiner and Norrie (Steiner and Norrie, 1997), which implemented a temporal ODB on top of O_2 .

No-overwrite strategies have been used in shadow-paging recovery strategies earlier, e.g., in System R (Gray et al., 1981), but with the limited buffer size at that time, the performance was not satisfactory. POSTGRES also employed a no-overwrite strategy (Stonebraker, 1987), but had also its performance problems, for several reasons, the most important being the buffer force strategy used.

The Vagabond approach is based on the same approach as log-structured file systems (LFS), which was introduced by Rosenblum and Ousterhout in (Rosenblum and Ousterhout, 1991). LFS has been used as the basis for two other object managers: the Texas persistent store (Singhal et al., 1992), and as a part of the Grasshopper operating system(Hulse and Dearle, 1996). Both object stores are page based, i.e., when an object has been modified, the whole page it resides on has to be written back, while the Vagabond approach is object based.

Some of the issues in this paper have also been previously presented in (Nørvåg, 2000b) and (Nørvåg, 2000c). A more in-depth study of issues related to temporal object database systems and more details about the Vagabond approach can be found in the author's doctoral thesis (Nørvåg, 2000e).

OBJECT MANAGEMENT ISSUES

In this section, we consider the physical management of objects on persistent storage, main memory management, clustering aspects, OID indexing, and temporal large objects.

Object Storage

A temporal database system can be implemented either through a *stratum* or an *integrated* approach. With the stratum approach, the database system is built on top of a non-temporal database system, and a layer converts temporal query language statements into conventional statements that are executed by the underlying system. Although this approach makes the introduction of temporal support into existing database systems easier, we do not see it as a long-term solution, because temporal query execution with this approach can be very costly.

Using an integrated approach, there are several way to organize the storage of objects. We will now discuss two interesting alternatives, the *partitioned storage* approach using in-place updating, and the *log-only* approach.

Partitioned Storage

Storage of data in a temporal database system is not very different from storage of data in a traditional database system. However, because current data tend to be more frequently accessed than historical data, the database is often partitioned into a *current store* and a *history store*. The two stores can utilize different storage formats, and even reside on different storage media (Ahn and Snodgrass, 1988). In this way, frequently accessed data is clustered together, stored on fast storage media, while historical versions can be stored on slower but cheaper storage media. In this way, the total storage cost is reduced, similar to the goal of general storage hierarchies.

One way to implement partitioned storage in a temporal ODB, is to store current version objects clustered together (similar to a non-temporal ODB), and write historical versions sequentially to an historical object store (for example a separate file). When an object is updated,

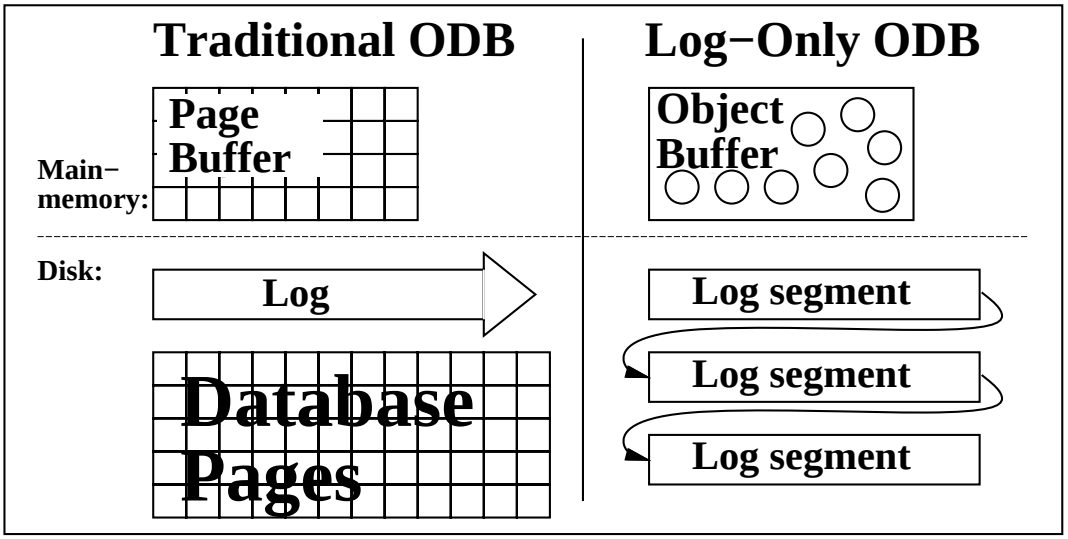

Figure 1: In-place update page server vs. log-only object database system.

the previous current version is copied to the history store before the new current version is update in-place. In order to be able to access the historical versions, a separate history index can be used. This index can be as simple as a B+-tree using the OID of the current version object, concatenated with time, as the index key. The leaf node entry is the OID of the current version of the object, the time interval where this version was valid, and the OID of the historical version. The location of the historical version is given through the OID in the leaf node. A variant of this approach has been used in the POST/C++ temporal object store (Suzuki and Kitagawa, 1996).

Log-Only Storage

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 update, but sooner or later, the update has to be done. This often results in the writing of lots of small objects, creating a write bottleneck. To avoid this, another approach is to eliminate the database completely, and use a *log-only* approach, based on the same philosophy as log-structured file systems, which was introduced by Rosenblum and Ousterhout (Rosenblum and Ousterhout, 1991). The log is written contiguously to the disk, in a no-overwrite way, in 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 possibly at the expense of read performance. Figure 1illustrates the most important differences between a traditional ODB, and a log-only ODB.

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 *we only use as much disk space as the size of the current version written.* In a system employing in-place updating, it is difficult to benefit from object compression, because the compression ratio will be different from version to version, and it is difficult to know how much space to reserve. Another

important advantage with the log-only approach is fast crash recovery. Only one pass through the log is necessary. This is very important in order to achieve high availability.

Logically, the log in a log-only system is an infinite length resource, but the physical disk size is, of course, not infinite. 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, in a temporal ODB most often old index nodes, will leave behind partially filled segments, the data in these near empty 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, we can get well clustered segments. In a traditional system using in-place updating, keeping old versions of objects, which is required in a transaction time temporal database system, usually means that the previous version has to be copied to a new place before update. This doubles the write cost. With the log-only approach, this is not necessary. Keeping old versions comes for free, except for the extra disk space.

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 version must be written to a new place. Thus, when supporting temporal data management, the indexing cost is the same in these two approaches.

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 pager 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. With bad clustering, main memory buffer utilization will be bad as well. A page based log-only ODB also makes transaction management difficult. To avoid page level locking, you essentially need to have 1) a separate log anyway, or 2) use ad-hoc techniques to solve the problem. Both solutions are likely to hurt performance and increase complexity, and have convinced us that an object based log-only ODB is the way to go. One of the objections against operating on object granularity, has been that the read cost will be prohibitively high. We have recently shown that this is not necessarily true for a log-only temporal ODB, and that with the workload we expect to be typical for temporal ODBs, the log-only temporal ODB is highly competitive with the traditional strategies (Nørvåg, 2000a). This is the strategy that is used in the Vagabond approach. It is important to note that one of the main reasons why previous approaches to log-only systems have not been able to achieve significant speedup compared to traditional systems, is that they have not tried to benefit from the “free object versioning” feature. It should also be noted that some of the problems in previous no-overwrite database systems have been solved in the Vagabond approach. For example, algorithms for steal/no-force buffer management, fuzzy checkpointing and fast commit have been developed.

Main Memory Management

In order to reduce disk I/O, the most recently used *index pages* are kept in an *index page buffer*. OIDX pages will in general have low locality, and in order to increase the probability of

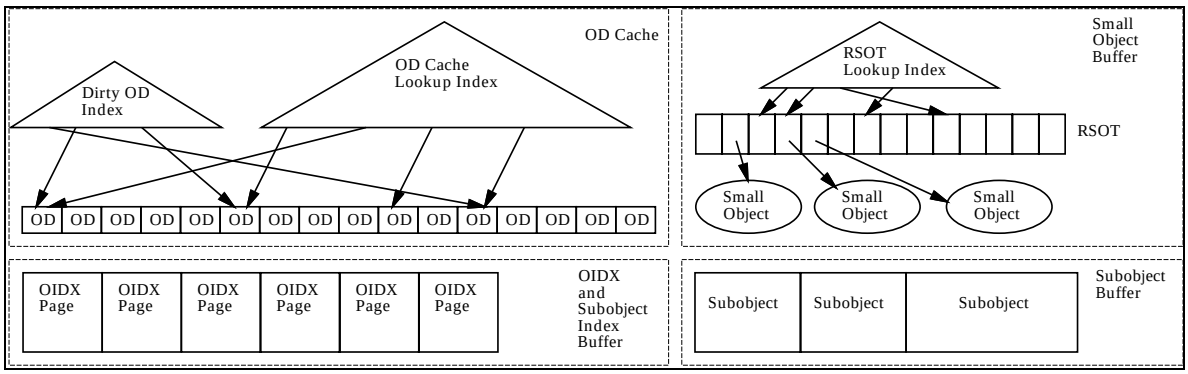

Figure 2: Important memory buffers.

finding a certain OD needed for a mapping from OID to physical address, it is also possible to keep the most recently used *index entries* (the ODs) in a separate *OD cache*, as is done in the Shore ODB (McAuliffe, 1997). With low locality on index pages, a separate OD cache utilizes memory better, space is not wasted on large pages where only small parts of them will be used. In non-temporal ODBs, an OD cache is only useful as a read buffer, because the OIDX update cost is low anyway (it is updated append-only). In a temporal ODB, on the other hand, individual entries are to be inserted into the OIDX. In order to make it possible to do the installation of ODs into the OIDX asynchronously, we also store these new ODs in the OD cache. At commit time, the new ODs are written to the log, and are inserted lazily into the OIDX later. By doing it this way, we increase the probability of having more than one entry for each index page that is updated. This significantly reduces the OIDX update cost.

In this paper, we describe the main memory buffers to be used in Vagabond (Nørvåg, 1999), a temporal ODB currently under development at the Norwegian University of Science and Technology. Special emphasis is given to the OD cache, small object buffer, and the OIDX page buffers. In order to make it possible for the system to adaptively change the size of the buffers, buffer models will be used to decide the buffer sizes.

We will in the following sections describe the most important main memory buffers in Vagabond, as illustrated in Figure 2: 1) OD cache, 2) small object buffer, and 3) large granularity buffers (for subobjects, OIDX-, and subobject index pages).

Object Descriptor Cache

As described previously, an OD cache is used to reduce the OIDX access costs, and reduces lookup costs as well as index update costs. ODs retrieved from lookups in the OIDX are inserted into the OD cache when retrieved, and new ODs resulting from object updates are inserted into the OD cache. However, new ODs from new objects are not initially inserted into the OD cache, they are written directly to the OIDX.

Designing an efficient OD cache is not straightforward. The requirements and functionality of the OD cache require a careful design. The entries in the OD cache are of a fine granularity, which means that additional overhead data can have a larger impact on performance than it would have in a page buffer, where the additional overhead usually is very small compared to the buffered items themselves. We will now describe operations the OD cache has to support, study

some aspects of the writeback of ODs to the OIDX, and then describe the architecture of the OD cache.

OD Cache Operations. The OD cache has to support the following operations:

- `lookup_current(OID)` Returns the OD of the current version of the object if the OD is in the OD cache.
- `lookup_most_recent(OID)` Returns the most recent OD that is resident in the OD cache.
- `lookup_at(OID,TIME)` Returns the OD of the object version valid at `TIME` if the OD is in the OD cache.
- `lookup_start(OID,TIME)` Returns the OD of the object version that has start time (commit time) `TIME` if the OD is in the OD cache.
- `lookup_end(OID,TIME)` Returns the OD of the object version that has an end time `TIME` if the OD is in the OD cache. The end time is equal to the start time of the next version of the object (or delete item if this was the last version before the object was deleted).
- `insert(OD)` Inserts OD into the OD cache. If this is a new current version of an object, set the end timestamp of the current version of the object if resident in the OD cache.
- `remove(OD)` Removes an OD from the OD cache.

In order to iterate through versions of an object, `lookup_at()` and subsequent `lookup_start()` operations can be used.

OD Cache-to-OIDX Writeback. Because of the size of each item in the OD cache, it is very important that when dirty (note that *dirty* in this context means *dirty with respect to the OIDX*, i.e., new or a modified ODs that have not yet been inserted into the OIDX. Persistent copies of the ODs have already been written together with the objects to the log.) ODs are to be written back to the OIDX, this can be done in batch. In order to reduce the number of index pages that has to be read (installation read of OIDX pages) and written, dirty ODs are sorted so that ODs that belong to the same OIDX pages can be installed into the OIDX pages at the same time. In most cases, disk seek time will also be reduced by updating the OIDX from the list of sorted ODs. This is similar to general use of an elevator algorithm when writing back pages from a page buffer.

A complicating factor for the OD cache, is the potentially large number of entries that has to be sorted. This can be time consuming. To have a sorted list of ODs, two approaches can be used:

1. When all dirty ODs from the last dirty list have been written back to the OIDX, a new list is generated by creating an array with pointers to all dirty ODs. This array is sorted, based on the OIDs, and then the ODs are asynchronously written back. The advantage with this approach, is that the extra space overhead is minimal. However, this approach has two important drawbacks:

a) ODs created after the array has been created and sorted, will have to wait until the next checkpoint interval, even if they belong to one of the OIDX pages that is retrieved and written when the array is processed.
b) The sorting of ODs can take several seconds of CPU time.

2. A *dirty OD index* with ordered elements, for example a binary tree, can be used. When a new OD is created, a pointer to the new OD is inserted into the index. During each checkpoint interval, the index is processed at least once. Because new entries are inserted immediately, we avoid the problem with the previous approach, where only ODs created during the previous checkpoint interval were available for the OD cache-to-OIDX writeback process. The disadvantage of this approach is a higher space overhead. For example, using a binary tree, two pointers are needed for each dirty OD. Note that a general priority queue is not sufficient for this index. The reason is that at some point in time, we have to be guaranteed that all entries inserted before a certain time (in this case, before the previous checkpoint interval), have been processed (in this case, before we can finish a new checkpoint).

We expect the space overhead of the dirty entry index to be compensated by increased writeback efficiency, and choose the dirty entry index approach.

OD Cache Architecture. It is possible to store *all* ODs, dirty as well as clean, in the index tree, and use the index tree as the access path for OD cache lookups as well. However, if that approach was used, we would have to scan through the whole index tree during each checkpoint interval. If most of the entries are clean, many CPU cycles will on average be needed in order to find the dirty ODs.

A better approach is to use one lookup index for all ODs in the OD cache, in addition to the dirty OD index. The lookup index is optimized for accesses to the OD of the most recent version of an object. The dirty entries are also indexed by this index, which means they are represented in both the dirty entry index and the lookup index. The reason for this is that without this redundancy, we would in many cases have to search both indexes when doing a lookup for a recent OD.

OD Cache Lookup Index. To understand the design of the OD cache index, it is important to remember that each update of an object creates a new OD. For each object, there will be one OD for each version, and more than one of these ODs can be in the OD cache at the same time. This means that even though the most frequent lookup operation is to retrieve the most recent OD of an object, it must be possible to store the other ODs in the OD cache as well, and it must be possible to retrieve these in an efficient way.

The index is based on a chained overflow hash table. The bucket to put an OD into, is chosen based on hashing the OID of the OD. In this way, all ODs of the same object (same OID) will be in the same bucket. The ODs of an object is inserted into a version tree, for example a binary tree, where time is used as the key. ODs with different OIDs can be hashed to the same bucket, and for each OID we have a separate version tree. The version trees are chained in a linear list. With an appropriate size of the hash table, the number of OIDs hashed to the same bucket should be low.

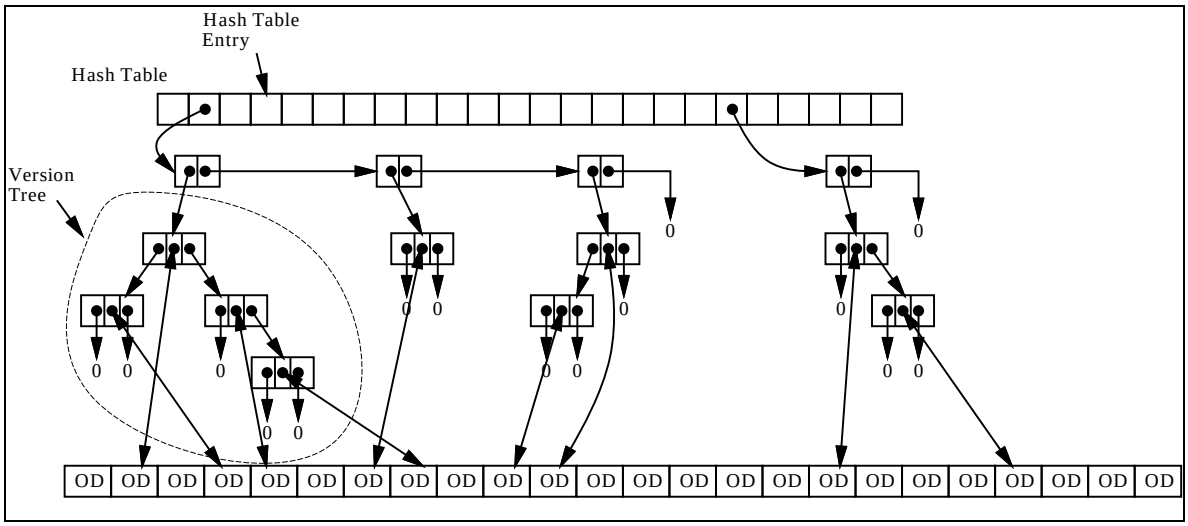

Figure 3: The OD cache lookup index.

The architecture of the OD cache lookup index is illustrated in Figure 3. Each entry in the hash table is a pointer to a list with pointers to the version trees. As can be observed, it would be possible to include the pointer to the next binary tree in the root of each version tree. In this way, we would avoid one pointer dereference. However, this is not done, because it could make some tree operations more complicated.

When choosing an appropriate version tree, the most important goals to achieve are 1) low insert cost, especially of a new current version OD, and 2) low lookup cost for the current version OD, which will be the most frequent operation. An ordinary binary tree is one possible solution. However, a problem with storing the ODs in a binary tree, is that if entries to be inserted into the tree have monotonically increasing key values, the result will be a linked list. Unfortunately, this is exactly the case when the inserts into the OD cache is ODs of new versions: The key value `TIME` is constantly increasing. One solution to this problem is to use a balanced tree, for example a splay-tree or a 2-3-tree. However, this increases the insert and space cost (it is possible to implement the splay tree with the same space cost as a binary tree, but this increases the access cost), and it is not certain that this approach will reduce the *average* access cost. Based on the knowledge of insert pattern and average number of versions, other heuristics can perform better, for example:

- When a new current version OD is inserted, its node is made the new root of the tree, and the current version of the tree is made the left subtree of this node. Non-current ODs are inserted into the tree following the binary tree insert algorithm. With this approach, search for the current version has a low cost.
- Another option is to keep a counter c which is increased for every insert of a new OD into the tree, and decreased for every delete from the tree (but always non-negative, i.e., if c is zero and we have a deletion, c will remain zero). If c reaches a certain threshold, the tree is reorganized and c is set to zero. Although a reorganizing approach in general is a bad idea, with higher cost than using a balanced tree, it can perform well in the OD cache if we assume that only a very few of the version trees have an insert rate that is high enough to

result in reorganizations. The space overhead is low for this approach, as we only need an additional counter for each version tree.

- On average, it is even possible that a list could perform well. The problem with this approach, is the high worst case cost (Rastogi et al. used a linear list for versions of data items in the Dalí main memory storage manager (Rastogi et). However, in Dalí, versioning is only used to support transient versioning, and not to provide support for temporal data. Hence, the length of a version list will usually be short.).

OD Cache Replacement. The OD cache will only have empty slots during startup, before enough objects have been accessed to fill up the OD cache. After the cache has filled up, one of the ODs resident in the OD cache has to be discarded before a new OD can be inserted. Only non-dirty ODs (with respect to the OIDX) can be discarded, and the clock algorithm is used as an LRU approximation to decide which of the candidate ODs should be discarded. The number of dirty ODs in the OD cache should be kept relatively low, to reduce the cost when searching for a candidate OD for replacement.

Small Object Buffer

In Vagabond, small objects are stored and retrieved as separate entities, and an object buffer is the only reasonable choice. (In a temporal ODB using in-place updating for current object versions, a dual buffer consisting of a combined object and page buffer can be used. In that case, the object buffer part can be implemented like the small object buffer described here.) Large objects have to be treated differently, because some of the subobject index pages and the subobjects of an object version might also be a part of other object versions. In this section aspects of the small object buffer are described, and in the next section buffering of large object subobject index pages and subobjects are described.

Modified Object Chain. For each active transaction, there is a *modified object chain*. This list contains the objects that have not yet been written to disk, but must be written before an commit operation can finish.

Small Object Buffer Architecture. For the objects in the small object buffer, a clock algorithm is used as an LRU approximation. The *resident small object table* (RSOT) is used to store administrative information on objects currently resident in the small object buffer. The access to the RSOT is through an index structure similar to the one used for lookups in the OD cache.

Although the information stored in the RSOT alternatively could be stored together with the ODs in the OD cache, the number of objects resident in memory is in general much smaller than the number of entries in the OD cache, making that approach less space efficient.

When an object is read into the buffer, its OD is removed from the OD cache, and reinserted into the OD cache when the object is discarded from the object buffer. Although this at first glance might seem to be inefficient, it simplifies the OD cache management considerably, and also has the benefit of removing interaction and synchronization between the OD cache and the small object buffer.

When a small object is retrieved, the memory location and the size of the object is inserted into the RSOT, together with the memory location and the size of the object. The reason for storing the object size in the RSOT entry, is that the object size in the OD is the size of the object while on disk. On disk the object might be compressed, and thus have a size different from the

size when in main memory. Using the physical location field in the OD to store the main memory location of the object could be done to save space, but if that was done, we would have a problem when the object was discarded from the buffer. We would then have lost the log address, and the OD would have to be discarded as well. That would make it necessary to do a costly OIDX lookup the next time the object was to be accessed.

Note that when an object is updated, a new OD is created for the object. If both versions are to be stored in main memory, a new RSOT entry has to be created for the new version.

Large Granularity Buffers

The most frequently used subobjects and OIDX and subobject index pages, are buffered in the large granularity buffers. In Figure 2, separate buffers are used for each of the categories, but it is possible to use a common buffer if desired. The large granularity buffers are similar to traditional disk page buffers, where an item is retrieved from disk to the buffer on demand.

Temporal Clustering and Access Patterns

The performance of page server based ODBs depend heavily on good clustering of objects, i.e., a high probability that more than one object on a disk page retrieved from the disk will be used in the near future, before the page is discarded from the buffer. A good clustering reduces the number of object pages that has to be read and written, and it also results in better buffer memory utilization. In practice, when different applications with different access patterns access the ODB, it is difficult to achieve good clustering. In a study by Tsangaris and Naughton (Tsangaris and Naughton, 1992), all practical clustering algorithms resulted in an average clustering less than 0.25. Although less of an issue when log-only storage is used (where we do not rely that much on clustering), writing related objects together increases the gain from prefetching and disk read ahead.

In non-temporal ODBs, clustering considers different objects, and we only try to predict cases like “when object O_i is accessed, it is also likely that object O_j will be accessed shortly after.” However, it is likely that in a temporal ODB application, a good object clustering includes *historical* object versions as well as *current* object versions (after all, the reason for storing the historical object versions is that we want to access them later!). A good example that illustrates this is that in a traditional ODB without support for temporal objects, we would often simulate object versions by including timestamp as an user managed attribute in the objects, and store the objects in an object collection. With temporal support, the user will only see one object, but can access the different object versions. Thus, even if the object versions are historical versions, it is possible that some of these will be part of the hot set of object versions. As a result, possible access patterns in a temporal ODB also include cases like:

- If the current version of object O_i is accessed, it is also likely that all the historical versions of object O_i will be accessed in the near future.
- If the current version of object O_i is accessed, it is also likely that the previous current version of object O_i will be accessed in the near future.
- If the current version of object O_i is accessed, it is also likely that all the historical versions of object O_j will be accessed in the near future.

- If the version of object O_i valid at time T_i is accessed, it is also likely that the version of object O_j valid at time T_i will be accessed in the near future.

Many other access patterns exist, but this illustrates the increased complexity that is introduced in the clustering process. It also shows that partitioned storage makes clustering more difficult, there is no simple and efficient way to include historical objects in the clustering together with current objects. If log-only storage is employed, clustering of temporal objects is much easier, as it facilitates adaptive reclustering during segment cleaning.

In a traditional system, it is possible for the user or database administrator to define some clustering strategy for a database, for example by defining a clustering tree or using clustering hints. These approach can also be extended to temporal ODBs, but necessitates continuous reordering, because it is impossible to reserve space for all new versions that are created. We expect that even if these explicit clustering techniques extended with time can be used, adaptive reorganization will be even more important in future systems.

OID Indexing

In a temporal ODB, it is necessary to use logical OIDs, and an OID index (OIDX) is needed to do the mapping from logical OID to physical location when retrieving an object. We aim at supporting temporal data, while still having current version access performance close to a non-temporal database system. This is difficult with the general multiversion access methods, e.g., the TSB-tree, the R-tree, or LHAM. Even if these index structures could give better support for temporal operations, we believe efficient non-temporal operations to be crucial, as they will probably still be the most frequently used operations. The use of general multiversion access methods will increase storage space considerably and index insert considerably, and many of the alternative methods will also incur a considerable degree of redundancy. TSB-trees and R-trees have both good support for time-key search. However, when indexing ODs, we use the OID as the key, and many queries will only be for perfect match, not OID range. One problem with LHAM is that it can have high lookup cost when the current version is to be searched for. As this will be a very frequently used operation, LHAM is not suitable for our purpose. Secondary indexes, on the other hand, will typically be realized from one of the traditional multiversion access methods mentioned above.

We have designed a new index structure, the Vagabond Temporal OID Index (VTOIDX). Our main goals in the design of the VTOIDX was:

1. Support for temporal data, while still having index performance close to a non-temporal/one version database system. Even if the use of other index structures could give better support for temporal operations, we believe efficient non-temporal operations to be crucial, as they will probably still be the most frequently used operations.
2. Efficient object-relational operation. This is achieved by the use of physical containers.
3. Easy tertiary storage migration of partitions of the index. An OIDX is space consuming, a size in the order of 20% of the size of the database itself is not unreasonable. In the case of migration of old versions of objects to tertiary storage, it is desirable, and in practice necessary, that parts of the OIDX itself can be migrated as well to avoid the need for large amounts of disk space for the OIDX of the migrated objects.

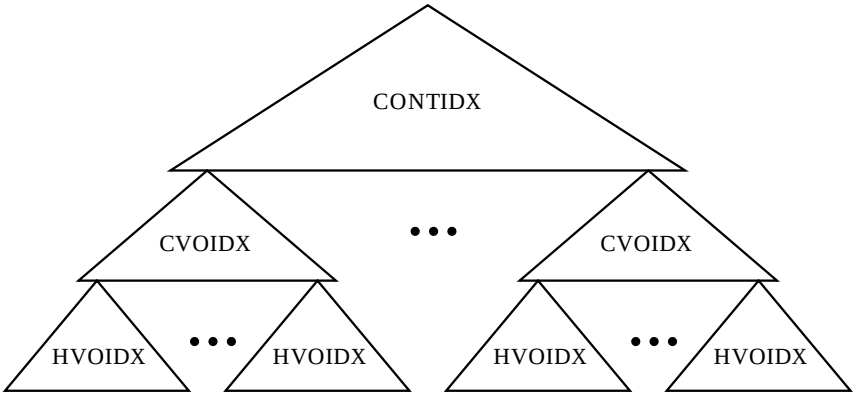

Figure 4: The Vagabond temporal OID index.

As a result, we ended up with an index structure that is an hierarchy of multi-way tree indexes, with three levels (Figure 4):

1. Container index (CONTIDX): The CONTIDX, one for each physical database, indexes the physical containers in a database. A physical container can be used, for example, to store all objects from a class, or for all objects in a collection. This can also be used to implicitly maintain class extents, and makes set based queries efficient. The pointers in the leaf nodes points to a current version OID index, one for each container.
2. Current version OID index (CVOIDX). The CVOIDX is an index for all ODs of the current versions of objects in one container, one OD for each object version.
3. Historical version subindex (HVOIDX): The CVOIDX itself is a nested tree index: The leaf nodes of the CVOIDX only contains the current version of the ODs, the previous versions are kept in separate subindex trees, which we call *historical version subindex* (HVOIDX). For each leaf node in the CVOIDX, there is a separate subindex tree, with non-current versions of all ODs that resides or have resided in the actual leaf node. Note that versioning is only needed on the bottom level of the hierarchy.

By having separate indexes for each container, it is easier to achieve high space utilization, because each subindex index is append-only. It is also easy to migrate a container to tertiary storage. By separating ODs of current and historical object versions, scan over the current version objects in a container is efficient. The CVOIDX and HVOIDX combination can be viewed as an extension of the Surrogate-Time (ST) index proposed by Gunadhi and Segev (Gunadhi and Segev, 1993). However, instead of one HVOIDX for each object, we use one HVOIDX for each CVOIDX node. The VTOIDX is described in more detail in (Nørvåg, 2000d).

Temporal Large Objects

In most ODBs, all objects smaller than a disk page is stored together with other small objects in a disk page, and objects larger than a disk page are segmented into “subobjects” and accessed through a large object index, a “subobject index.” In a log-only database system like Vagabond, we can use a more flexible approach, where the threshold for when to consider the object as a

large object can be much smaller than one page, for example 512 bytes, or if desired, much larger, for example 128 KB.

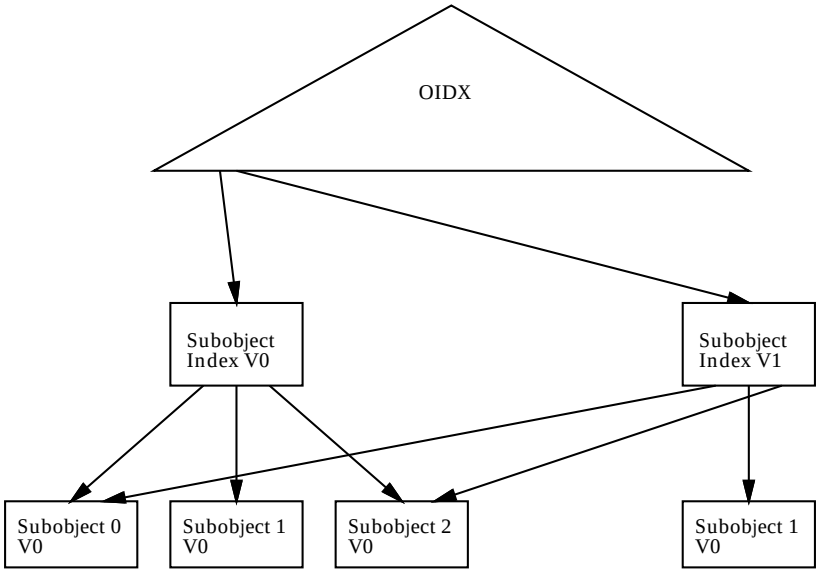

Figure 5: Versioned large object. V0 denotes the first version, V1 denotes the second version.

EXODUS Large Objects

There are several ways to manage the subobjects in a temporal ODB, but it should satisfy one important requirement: *storage efficient versioning*, i.e., when a large object is updated, as few subobjects as possible and as few of the subobject index nodes as possible should be rewritten. This can be achieved by the use of a subobject index that also takes care of versioning, similar to the EXODUS large storage objects (Carey et al., 1986).

The EXODUS storage system supports versioning, as illustrated in Figure 5. On top of the figure is the OIDX, which has separate ODs for each version of an object (in EXODUS, physical OIDs are used, so there is not actually an OIDX). To the left we have the initial version of an indexed large object. When parts of the object is updated, only the updated subobjects are stored. The index structure for the new version points to the previous version of the subobjects for those parts that have not been modified. This versioning also applies to the large object indexes themselves. If the indexes have more than one level, only modified parts of the index are rewritten. The result is minimal duplication of subobjects, efficient update of the subobject index, and efficient access to parts of the large objects.

Vagabond Large Objects

A Vagabond large object is based on the EXODUS large storage objects. To make large objects flexible, we have extended the EXODUS large storage object:

- Vagabond, the subobject threshold and subobject size can be different for different object classes. This is very useful, because different object classes can have different object

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

- In the internal large object index nodes, we use `(NavigDesc,pointer)` tuples instead of `(count,pointer)` tuples, where the `NavigDesc` (Navigational Descriptor) is a variable length structure. In this way, a Vagabond large object is a generalization of the EXODUS large storage objects. EXODUS large storage objects can be realized by using a counter as the `NavigDesc`, this is the default case.

More complex indexes and other special objects can be implemented as more general Vagabond large objects by using more complex NavigDescs. For example, R-trees can be realized by using rectangle descriptors in the `NavigDesc` data structure, a `(timestamp,key)` tuple can be used for temporal indexes etc.

OBJECT ACCESS AND QUERIES

In non-temporal ODBMSs, ODMG's OQL or similar query languages can be used for ad-hoc queries. Similar to the way OQL is a superset of the part of standard SQL that deals with databases queries, it is possible to design a temporal OQL that is a superset of TSQL2. One such approach has been described by Fegaras and Elmasri (Fegaras and Elmasri, 1998). However, one of the main advantages of ODBMSs is the avoidance of the language mismatch by providing computationally complete data manipulation languages with no mismatch between language and storage. In the ODMG standard, language bindings based on C++, Java and Smalltalk are described. Such language bindings are also needed for temporal ODBMSs. It should also be noted that in order to use methods in queries, these issues have to be resolved.

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

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

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

Temporal C++ Binding

In this section, we describe two approaches that extend the C++ language binding with support for access to historical data in a transaction-time ODBMS. The first approach is based on the language binding used in POST/C++ (Suzuki and Kitagawa, 1996), while the second is to our knowledge new. The concepts of these approaches can also be employed for a Java language binding.

Explicit Object Version Access

The easiest way to integrate object version access into the programming language is to provide explicit access to the versions. This is the way it is done in POST/C++ (Suzuki and Kitagawa, 1996). Given an OID, the program can be given a pointer to a historical version valid at a particular time by calling a function `snapshot(OID,time)`. It is also possible to create iterators that can be used to navigate the versions of an object in chronological sequence.

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

The Explicit Snapshot Approach

A better and "cleaner" alternative than the one described above is to use explicit snapshots. Before calling a method in a historical version current at time `ts`, we set the snapshot time with a call to the function `set_snapshot(d_Timestamp ts)`. After the `set_snapshot()` function has been called, an access to a particular object will be to the object version current at time `ts`, *even though the reference is through a* `d_Ref` (A `d_Ref` is a reference to an object.) A call to `set_current()` will set accesses back to normal, i.e., an access to a particular object will be to the current object version. Methods called in historical objects should in general be immutable, i.e., read-only methods. The advantage of this approach is that all object versions accessed will be object versions valid at the same time.

All access, creation, modification and deletion of persistent objects must be done within a transaction. In the ODMG C++ binding, transactions are implemented as objects of the class `d_Transaction` (Cattell, 1997):

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

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

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

  private:
     ...
  };
```

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

To Bind or not to Bind?

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

CONCLUSIONS

The area of temporal ODBs is still immature, and there are many design issues that need to be solved. In this paper, we have discussed problems and possible solutions derived from the design of Vagabond. Our most important observations are:

- The properties of the log-only approach, together with results from performance analysis of the approach, are very convincing.
- In a database system, efficient use of main memory for buffering is important. In a temporal ODB, object indexing is a possible bottleneck. Hence, efficient buffering of index nodes as well as index entries is crucial for performance. In this paper, we have described the design of suitable main memory buffer, and described in detail the management of fine-granularity buffer items like OID index entries and small objects.
- Language bindings in a temporal ODB is a still unchartered area. It is not obvious whether traditional language bindings or queries should be used in order to access historical data.
- Clustering and access patterns is possibly even more important in temporal ODBs than in traditional ODBs. This is an important area that should be studied further.

REFERENCES

Aguilera, V., Cluet, S., Veltri, P., Vodislav, D., and Wattez, F. (2000). Querying XML documents in Xyleme. Technical Report 182, Verso/INRIA.

Ahn, I. and Snodgrass, R. (1988). Partitioned storage for temporal databases. *Information Systems*, 13(4).

Carey, M., DeWitt, D., Richardson, J., and Shekita, E. (1986). Object and file management in the EXODUS extensible database system. In *Proceedings of the 12th VLDB Conference*.

Cattell, R., editor (1997). *The Object Database Standard: ODMG-93. Release 2.0*. Morgan Kaufmann.

Fegaras, L. and Elmasri, R. (1998). A temporal object query language. In *Proceedings of the Fifth International Workshop on Temporal Representation and Reasoning*.

Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., and Traiger, I. (1981). The recovery manager of the System R database manager. *ACM Computing Surveys*, 13(2).

Gunadhi, H. and Segev, A. (1993). Efficient indexing methods for temporal relations. *IEEE Transactions on Knowledge and Data Engineering*, 5(3).

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

McAuliffe, M. L. (1997). *Storage Management Methods for Object Database Systems*. PhD thesis, University of Wisconsin-Madison.

Nørvåg, K. (2000a). A comparative study of log-only and in-place update based temporal object database systems. In *Proceedings of the Ninth International Conference on Information and Knowledge Management (CIKM'2000).*

Nørvåg, K. (2000b). Design issues in transaction-time temporal object database systems. In *Proceedings of ADBIS-DASFAA'2000.*

Nørvåg, K. (2000c). Main-memory management in temporal object database systems. In *Proceedings of ADBIS-DASFAA'2000*.

Nørvåg, K. (2000d). The Vagabond temporal OID index: An index structure for OID indexing in temporal object database systems. In *Proceedings of the 2000 International Database Engineering and Applications Symposium (IDEAS).*

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

Nørvåg, K. (1999). The Vagabond parallel temporal object-oriented database system: Versatile support for future applications. In *Proceedings of Norsk Informatikkonferanse 1999.*

Rastogi, R., Seshadri, S., Bohannon, P., Leinbaugh, D., Silberschatz, A., and Sudarshan, S. (1997). Logical and physical versioning in main memory databases. In *Proceedings of the 23rd VLDB Conference.*

Rosenblum, M. and Ousterhout, J. K. (1991). The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*.

Singhal, V., Kakkad, S., and Wilson, P. (1992). Texas: An efficient, portable persistent store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*.

Steiner, A. and Norrie, M. C. (1997). Implementing temporal databases in object-oriented systems. In *Proceedings of the 5th International Conference on Database Systems for Advanced Applications (DASFAA'97).*

Stonebraker, M. (1987). The design of the POSTGRES storage system. In *Proceedings of the 13th VLDB Conference.*

Suzuki, T. and Kitagawa, H. (1996). Development and performance analysis of a temporal persistent object store POST/C++. In *Proceedings of the 7th Australasian Database Conference*.

Tsangaris, M. and Naughton, J. (1992). On the performance of object clustering techniques. In *Proceedings of SIGMOD'92*.

Kjetil Nørvåg is an Associate Professor in the Department of Computer and Information Science at the Norwegian University of Science and Technology. He received a Dr. Ing. degree in computer science from the Norwegian University of Science and Technology in 2000, and was a postdoctoral researcher in the VERSO group at INRIA (France) in 2001. His major research interests include query and storage of XML documents, object database systems, temporal database systems, query processing, and access methods.