

# The Vagabond Temporal OID Index: An Index Structure for OID Indexing in Temporal Object Database Systems

Kjetil Nørnvåg

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

## Abstract

*In an object database system using logical OIDs, an OID index (OIDX) is necessary to map from logical OID to the physical location of an object. In a temporal object database system (TODB), this OIDX also contains the timestamps of the object versions. OIDX maintenance can be very costly, and can easily become the bottleneck of such a system. The main reason for this, is that in a TODB the OIDX needs to be updated every time an object is updated. In order to reduce the access costs, a new index structure, particularly suitable to TODB requirements, is necessary. In this paper, we describe an OIDX for TODBs, the Vagabond Temporal OID Index (VTOIDX). The main goals of the VTOIDX are 1) support for temporal data, while still having index performance close to a non-temporal (one-version) database system, 2) efficient object-relational operation, and 3) flexible tertiary storage migration of partitions of the index. In this paper, we describe the physical organization and the operations of the VTOIDX.*

## 1 Introduction

In a temporal object database system (TODB), object updates do not make previous versions inaccessible. On the contrary, previous versions of objects can still be accessed and queried. Temporal databases can either support *transaction time*, *valid time*, or both. In a transaction-time TODB, which is the context of this paper, a system maintained timestamp is associated with every object version. This timestamp is the commit time of the transaction that created this version of the object. In valid-time database systems, a time interval is associated with every object, denoting the time interval which the object is valid in the modeled world.

An object in an object database system is uniquely iden-

tified by an object identifier (OID), which is also used as a “key” when retrieving an object from disk. OIDs can be *physical* or *logical*. If physical OIDs are used, the disk block where an object resides is given directly from the OID, if logical OIDs are used, it is necessary to use an OID index (OIDX) to map from logical OID to the physical location of the object. Most of the early ODBs and storage managers used physical OIDs because of its performance benefits, and many of the commercial ODBs still do. However, physical OIDs have some major drawbacks: relocation, migration of objects, and schema changes are more difficult. In this paper, we assume that logical OIDs are used.

In a TODB, it is usually assumed that most accesses will be to the current versions<sup>1</sup> of the objects in the database. In order to keep these accesses as efficient as possible, and benefit from object clustering, the database is partitioned. The current version objects are stored in the *current database*, and the historical (previous) versions are stored in the *historical database*. When an object is updated in a TODB, the previous current version is first moved to the historical database, before the new version is stored in-place in the current database. The OIDX needs to be updated *every time an object is updated* (but note that as long as the OID/timestamp/location records are written to the log before commit, we do not need to update the OIDX itself immediately).

We have in a previous paper [10] studied OIDX performance, and have shown that OIDX maintenance can be quite costly, especially when updating objects. Even if the use of index entry caching in main memory [10] and on persistent storage [8] can be used to reduce the access cost, a new index structure is necessary, especially suitable to the TODB requirements. Such an index structure, which has been developed in the context of the Vagabond TODB [9], will be described in this paper.

---

<sup>1</sup>The current version of an object is the most recent version of a non-deleted object.

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 how multiversion indexing can be done efficiently in TODBs. In Section 4 we describe the Vagabond Temporal OIDX (VTOIDX) in detail, including an overview of the physical data organization and algorithms. Finally, in Section 5, we conclude the paper.

## 2 Related work

OID indexing alternatives in traditional, non-temporal, ODBs has been studied by Eickler et al. [1]. We have in a previous paper developed a cost model of OIDX lookup cost in TODBs, and studied how memory can be best utilized in buffering of OIDX pages and index entries [10]. In this case, a temporal OIDX which was a simple extension of a traditional OIDX was assumed.

There have also been much work on multiversion access methods and secondary indexing of temporal data, for example using a TSB-tree [6], R-tree [5],<sup>2</sup> or LHAM [7]. However, as will be shown later in this paper, traditional multiversion access methods are not suitable for OID indexing in TODBs. To our knowledge, the only other paper discussing the issue of a temporal OIDX is the presentation of the POST/C++ temporal object store [13].

## 3 Multiversion indexing

The entries in the OIDX are called *object descriptors* (OD). The ODs contain the necessary information to map from OID to physical location. In Vagabond, we use one object descriptor (OD) for each version of an object, and an OD also includes the timestamp of the actual object version. The index structure has to support access to ODs of current as well as historical versions of the objects. Before we present our solution to multiversion indexing in the next section, we will describe the use of physical containers, take a look on some characteristics of OIDs and OID search, and discuss different multiversioning alternatives.

### 3.1 Physical object containers

Performance can be improved considerably if index entries from objects that are accessed together close in time, are clustered together on the same index nodes. In some cases, the access pattern will be close to the object creation pattern. However, this can not be relied on.

In many page server ODBs, the objects are stored in containers (also called files). Which container to put an object

<sup>2</sup>The R-tree is a spatial access method, but can also be used as a temporal index by indexing keys (OIDs in TODBs) in one dimension, and time in the other dimension.

into, is decided when the object is created, and part of the OID is used to identify the container where the object is stored. In many systems, it is possible to define clustering trees that can be used by the systems as a basis for the clustering decision, for example clustering together objects that are likely to be accessed together, and members of a set that are later going to be accessed in scan operations. A similar approach is used in our indexing structure. Similar to object clustering in page servers, which reduces the number of pages to read and update, clustering together related OIDs will reduce the cost of index accesses.

All objects in a database are members of one physical container. The container an object belongs to is encoded into the OID, and as a consequence, forwarding must be used if migration is desired. This will imply an extra lookup for each access to an object that has been migrated to a new container.

Given a certain size of an OID, using a fixed part of the OID as a container identifier reduces the number of bits to represent the unique number of an object. As a consequence, the number of objects that can exist is reduced. To avoid this problem, it is possible to increase the size of the OID compared to the size used if index clustering is not employed. This imply an extra cost, but this cost is cheap compared to the alternative of *not* using the container approach. A larger OID will make objects with object references larger, but access cost can be reduced, because in most cases, a smaller number of index nodes need to be retrieved. The reduced OIDX update cost will significantly increase the throughput.

The containers can also be used to realize logical collections, for example sets (relations), bags or class extents.<sup>3</sup> Scan and query against collections can then be done efficiently. It is important to note that in other ODBs, where a physical OID is used or the OIDX has no support for containers, maintaining class extents can be costly.

It is also interesting to note that the use of containers gives us more flexibility in deciding the length of the search path for objects. It is possible to store hot spot objects into small containers to get a short search path.

### 3.2 Characteristics of OIDs and OID search

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

- If we assume the unique part of an OID to be an integer, new OIDs are in general assigned monotonic increasing values. In this case, there will never be inserts of new key (OID) values between existing keys (OIDs).

<sup>3</sup>A class extent is a collection of all the objects of a certain class in a database.

- As a result, the keys in the index, the OIDs, are not uniformly distributed over a domain as keys commonly are assumed to be.
- If an object is deleted, its OID will never be reused.

In a non-temporal (one-version) OIDX, the entries in the OIDX are seldom updated, and removal of entries belonging to objects that have been deleted can be done in batch and/or as a background activity. If using a tree based OIDX, new entries will be added append-only. By combining the knowledge of the OIDX properties and using tuned splitting, an index space utilization close to 1.0 can be achieved. If something similar to container clustering is used, however, inserts could be needed, and space utilization would decrease. This can be avoided by using a hierarchy of multi-way tree indexes, as will be shown later.

Index accesses will mostly be for perfect match, there will be no key range (in this case a range of OIDs) search. (With container clustering, we will also have OID range accesses. However, accessing objects in a container will often result in additional navigational accesses to referenced objects.)

It is important to remember that there will in general be no correlation between OID and object key, so that an ordinary object key range search will not imply an OID range search in the OIDX. If value based range searches on keys (or other attributes in objects) are frequent, additional secondary indexes should be employed, for example B+-trees or temporal secondary indexes.

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

We will in the following subsections study several alternative ways to organize the temporal OID indexing, and discuss advantages and disadvantages for each of the following alternatives:

1. One index structure, with all ODs, current as well as historical versions.
2. One index structure for current ODs, with links to the historical versions.
3. Nested tree index, one index with version subindexes.
4. Two separate index structures, one for current ODs, and one for historical ODs.

### 3.3 One index structure

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

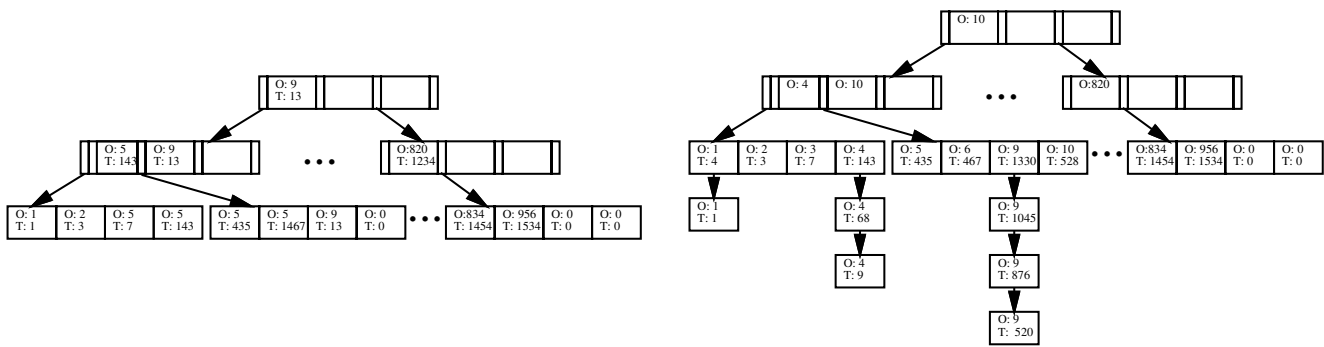
**Composite index.** With this alternative, we use the concatenation of OID and commit time,  $OID||TIME$  as the index key, as illustrated to the left in Figure 1. By doing this, the ODs of the different versions of an object will be clustered together in the leaf nodes, sorted on commit time. As a result, search for the OD of the current version of a particular object as well as retrieval of ODs for versions of one particular object created during a particular time interval can be done efficiently.

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

1. Even in an index organized in physical containers, leaf nodes will contain a mix of current and historical ODs. The ODs of current versions are not clustered together, something that makes a scan over the ODs of current versions inefficient.
2. An OIDX is space consuming, a size in the order of 20% of the size of the database itself not being 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. This is difficult when current and historical versions reside on the same leaf pages.

The composite index is used in the POST/C++ temporal object store [13] (based on the Texas object store [11]). In POST/C++, objects are indexed with physical OIDs, and a variant of the composite index structure is used to index historical versions. Because of the use of physical OIDs, when an object is updated in POST/C++, a new object is created to hold the previous version. After the previous version has been copied into the new object, the new version is stored where the previous object had previously resided. A positive side effect of doing it this way, is that current and historical object versions are separated.

**Use of general multiversion access methods.** Using general multiversion access methods, for example a TSB-tree



**Figure 1. One-index structure. To the left, a composite index using the concatenation of OID and commit time, as the index key, and to the right, an index using version linking.**

[6], R-tree [5], or LHAM [7], is also an alternative. However, LHAM is of little use for OID indexing, because it can have a high lookup cost when the current version is to be searched for. As this will be a very frequently used operation, LHAM is not suitable for our purpose.

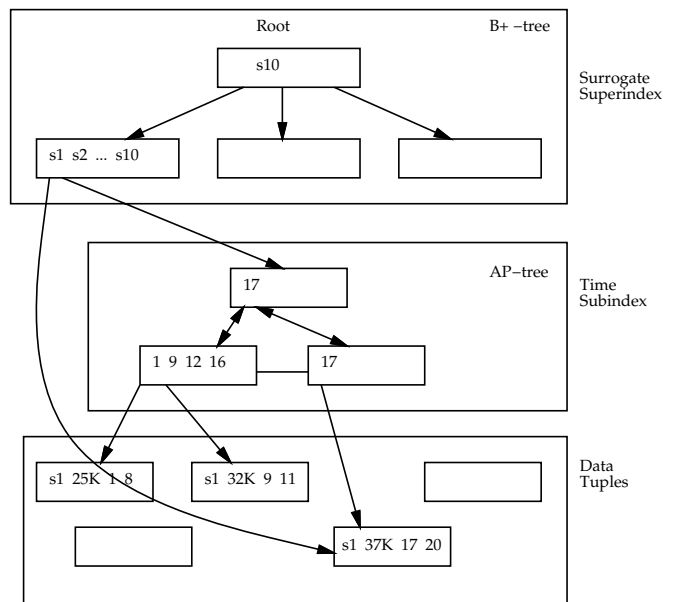
TSB-trees and R-trees have both good support for time-key range search, and make index partitioning possible. However, when indexing ODs, most queries will be OID lookups, and when OID is the key, support for key range search is of little use. Even if the use of TSB- or R-trees could give better support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

### 3.4 Index with version linking

To avoid the disadvantages of the previous index alternative, only the ODs of the current versions of the objects are kept in the index. Each OD in the index has a list of ODs of the historical versions, as illustrated to the right in Figure 1, and the ODs of the historical versions are kept in this list.

To reduce access costs for historical versions, it is possible to link the object versions instead of the ODs. This has similarities with the approach used in POSTGRES [12], where a link exists from one version of a tuple to the next (in POSTGRES the list started with the oldest version, so that in order to retrieve the current version the whole list had to be traversed).

A linked list approach, whether it is the ODs or the objects that are linked, has some serious disadvantages. For all



**Figure 2. Nested ST indexing [4].**

operations on temporal versions, the list must be traversed, resulting in extra disk accesses.

### 3.5 Nested tree index: index with version subindexes

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

An example of a nested tree index is the Surrogate-Time (ST) index [4], illustrated in Figure 2. The *surrogate superindex* indexes the key values of the tuples, and is implemented with a B+ tree. Each leaf node entry has a direct

pointer to the current data tuple, as well as a pointer to a *time subindex*. The *time subindex* is an append-only tree, with time as the key value. Each entry in the subindex has a pointer to the data tuple with the timestamp in the key of the entry.

### 3.6 Separate indexes for current and historical versions

In order to make read accesses to current version as efficient as possible, one index for ODs of current versions of objects can be used, and a separate index for ODs of historical versions. The index for the historical data can be organized as either of the three previous index organizations.

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

## 4 VTOIDX: The Vagabond Temporal OIDX

In Vagabond, a container identifier is included in the object identifier, which is composed of three parts:

1. Server group identifier (*SGID*), which is the identifier of the server where the object was created. This is only used in a distributed system.
2. Container Identifier (*CONTID*), which identifies the physical container the object belongs to.
3. Unique serial number (*USN*). Each object created on a particular server *SGID* and to be included in container *CONTID* gets a *USN* which is one larger than the previous *USN* allocated.

Our main goals in the design of the Vagabond Temporal OIDX (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 still be the most frequent operations.

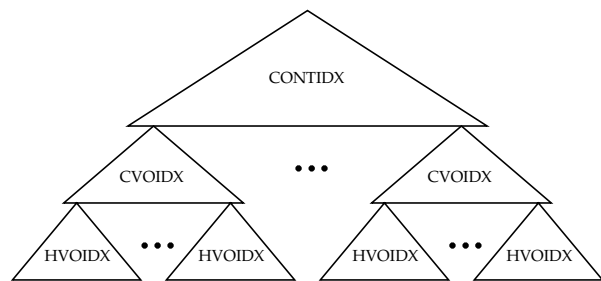


Figure 3. The Vagabond temporal OIDX.

2. Efficient object-relational operation. This is achieved by the use of physical containers, which is described below.
3. Flexible tertiary storage migration of partitions of the index.

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

### 4.1 VTOIDX physical data organization

Based on the analysis of the different OIDX alternatives, we have designed the VTOIDX. The VTOIDX is an hierarchy of multi-way tree indexes, with three levels, as illustrated in Figure 3:

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

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

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

using prefix compression of the OID in the index entries). This increases fan-out as well as the number of ODs in the leaf nodes. As long as the upper levels of the tree are buffered, the benefits of more ODs in a leaf node outweighs the extra cost of the higher number of levels.

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

**Container index.** There is one CONTIDX for each database, and it indexes the physical containers in the database. The pointers in the leaf nodes points to a current version OIDX, one for each container. The entries in internal nodes as well as leaf nodes are  $(CONTID, pointer)$  tuples.

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

**Current version OIDX.** There is one CVOIDX for each container, and it indexes the ODs of all the current versions of the objects in the container. The CVOIDX and HVOIDX combination is based on the ST index (see Section 3.5), and the CVOIDX itself is similar to the surrogate superindex in the ST index.

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

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

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

**Historical version OIDX.** For each leaf node in the CVOIDX, there is a separate HVOIDX subindex tree, with ODs of the non-current versions of objects that reside or

have resided in the actual leaf node. The HVOIDX is similar to the subindex in the ST index (see Section 3.5), but instead of using one subindex for each key value as in the original ST subindexes, several objects share one subindex in our index. The subindex uses the concatenation of OID and commit time,  $OID||TIME$ , as the key. In this case, we have efficient access to the ODs of a particular object, which will be clustered together, and at the same time have clustered ODs of current versions in a container.

In the HVOIDX trees, the concatenation of *USN* and commit time,  $USN||TIME$ , is used as the index key during insert and search in the tree. Entries in the internal nodes of a HVOIDX tree are  $(USN||TIME, pointer)$  tuples. The leaf nodes contain the ODs only, because each OD contains *USN* as well as the timestamp.

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

When each CVOIDX leaf node has one HVOIDX subtree, it is possible that some HVOIDX subtrees only have a very few entries. To optimize space usage, several CVOIDX leaf nodes could share one HVOIDX subtree. However, this would give each HVOIDX root node more than one parent, and each time the HVOIDX was updated each of these have to be updated, which increases the insert cost. We do not think this will be beneficial. We believe the subindex caching introduced below will reduce the need for shared HVOIDXs, and it is also very likely that most members of a container will have the same versioning characteristics. In general, we expect the space utilization to be acceptable even if sharing is not used.

**Subindex caching.** When a temporal object is updated, a new OD is created, and the old one pushed down into an HVOIDX. As a result, both the leaf node in the HVOIDX as well as a leaf node in the CVOIDX has to be updated (plus internal nodes in the case of node splits). To reduce the number of nodes to rewrite, and a corresponding number of installation reads, the ODs of the most recent historical versions are stored in the leaf nodes of the CVOIDX. We

call this technique *subindex caching*.

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

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

**Comments on the index trees.** Many of the insert operations in the VTOIDX will actually be append operations. In the standard B<sup>+</sup>-tree insert algorithm, contents in a split node are distributed over the old and new node. If entries are only appended to the index, this would result in a tree with only 50% space utilization. To avoid this, we use *tuned splitting*, a technique also used in the Monotonic B<sup>+</sup>-tree [2] for the same purpose. When tuned splitting is used, entries are not distributed evenly over the old and the new node when a node is split, only the new entry is stored in the new node.

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

## 4.2 Operations on the VTOIDX

In this section we describe the most important operations on the VTOIDX, which are done as a result of container and object operations.

**Creating or deleting containers.** Creating a new container is done by inserting a new entry into the CONTIDX.

The value of a new *CONTID* will always be larger than existing *CONTIDs*, so this will actually be an append operation, and tuned splitting is used to achieve high space utilization.

Physically deleting a container is done by deleting the container entry in the CONTIDX. This operation should be done after the corresponding CVOIDX and HVOIDX indexes have been deleted.

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

1. Require the application code to do some kind of exception handling when a temporal query tries to access a deleted container.
2. Keep the CVOIDX and associated HVOIDXs in the system, but flag all update attempts as errors.

**Search for current object version.** Search for the OD of the current version of an object is done by first using the *CONTID*, which is a part of the *OID*, to do a lookup in the CONTIDX to get a pointer to the CONTIDX where the OD resides in. When the CONTIDX root node has been retrieved, the *USN* of the *OID* is used to search the CVOIDX, and if the object with the actual *OID* exists and is valid, it will be found in a CVOIDX leaf node. For both searches, the standard B<sup>+</sup>-tree search algorithm is used.

**Create new object.** When a new object is created, the application that created the object decides which container the object (and its OD) should reside in, and a new *OID* is allocated.

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

**Update temporal object.** When an object is updated, the OD for the new version has to be inserted into the tree. The first step is to find the CVOIDX leaf node where the current version of the OD is stored.

In the case of a non-temporal object, the old OD is simply replaced with the new one. In the case of a temporal object, the old OD is replaced with the new OD, and the old OD is inserted into the HVOIDX subindex where the historical versions are kept. While the use of the *USN* only is used in key comparison until this point, when inserts are to be done into the HVOIDX, the concatenation of *USN* and commit time, *USN||TIME*, is used as the HVOIDX index key. Note that in this case, we have also inserts into the tree, and not only append operations. Therefore, we use the standard B<sup>+</sup>-tree insert algorithm in this case, without employing tuned split. To reduce the average update cost, we also employ subindex caching as described previously. In this case, when we push down ODs to the HVOIDX, we have more than one OD to insert, and the average cost for each OD is reduced.

**Delete object.** In the case of a non-temporal object, the OD is simply removed from the actual CVOIDX leaf node where it resides.

In the case of a temporal object, the current OD is moved from the CVOIDX to the HVOIDX, and an additional tombstone OD is inserted into the HVOIDX subindex (the tombstone OD is an OD where physical location is NULL, and the timestamp is the commit time of the transaction that deleted it).

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

**Vacuuming.** Even though storage cost is decreasing, storing an ever growing database can still be too costly for many application areas. A large database can also slow down the speed of the database system by increasing the height of index trees (even though this can be avoided with multi level indexes, at the cost of a more complex system). As a consequence, it is desirable to be able to physically delete data which has been logically deleted, and non-current versions of data that is not deleted. This is called *vacuuming* (but note that vacuuming is also sometimes used as another term for the migration of historical data from secondary storage to tertiary storage).

When object versions are vacuumed, their ODs residing in the HVOIDX will be deleted. This is done according to

the standard B<sup>+</sup>-tree delete algorithm.

**Search for object version valid at time  $t_i$ .** First, a search is done to find the CVOIDX leaf node where the OD of the current version of the object resides. If the timestamp of this OD is less than  $t_i$ , this OD is the result of the search. If not, the HVOIDX is searched to find the OD of this object that have the *largest timestamp less than  $t_i$* . Note that the ODs of deleted objects only reside in the HVOIDX. Thus, even if an OD with the actual OID is not found in an the CVOIDX leaf node, we still have to search the HVOIDX if we do not get a match in the CVOIDX leaf node.

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

**Search for current version of all objects in a container.** This is the traditional scan operation. In the VTOIDX, this is done in the same way as in a traditional B<sup>+</sup>-tree, by returning the entries in the CVOIDX leaf nodes.

**Search for all versions of an object.** This operation is done by first retrieving the CVOIDX leaf node where the OD of the current version of the object resides, and then retrieve the ODs of all versions of this object from the corresponding HVOIDX.

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

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

If this type of query is expected to be frequent, it would be beneficial to keep the tombstones ODs of deleted objects in the CVOIDX leaf nodes. If this is done, we could avoid further searches in the HVOIDX if all objects represented by the particular CVOIDX leaf node was deleted before time  $t_i$ . In that case, we know that they could not be valid at time  $t_i$ . As we do not know for sure the frequencies

of different query types in future systems, it is difficult to say if this kind of query will be frequent enough to justify keeping tombstones of deleted objects in the CVOIDX leaf nodes.

**Update all objects in a container.** Only objects that are still valid can be updated, so this operation is essentially:

1. Retrieve the ODs of all current objects in the container.
2. Each object update creates a new OD to be inserted into the VTOIDX. When inserting ODs into the container in this way, it will be done very efficiently.

**Migration to tertiary storage.** Any subtree of the VTOIDX can be migrated to tertiary storage. All nodes in all levels of the VTOIDX are addressed by a 64 bit logical location address, which is used to select storage device and location on the actual storage device.

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

### 4.3 Concurrency control aspects

Ordinary tree locking algorithms can be used to control access to the tree. Note also that during normal processing, only ODs of non-temporal objects are modified. When a new version of a temporal object is created, a new OD is created and inserted into the tree.

In traditional systems, leaf nodes are usually linked together. This can be used to make some of the B<sup>+</sup>-tree operations more efficient and improve concurrency in the VTOIDX.

## 5 Conclusions

OID indexing in TODBs poses great challenges. Because of the update costs, it can easily become the bottleneck in such systems. Previous studies of OID indexing in TODBs have shown that achieving acceptable performance can be difficult if most of the OIDX does not fit in main memory. In this paper, we have described an index structure, the VTOIDX, that should perform well, even in systems where the OIDX is much larger than the available

main memory buffer. The VTOIDX should also be capable of fulfilling the goal of OIDX lookup performance close to conventional systems on current data, good performance on object-relational operations, and flexible tertiary storage migration, which will be important for future TODBs.

## References

- [1] A. Eickler, C. A. Gerlhof, and D. Kossmann. Performance evaluation of OID mapping techniques. In *Proceedings of the 21st VLDB Conference*, 1995.
- [2] R. Elmasri, G. T. J. Wu, and V. Kouramajian. The time index and the monotonic B<sup>+</sup>-tree. In A. U. Tansel, J. Clifford, S. K. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal databases: theory, design and implementation*. The Benjamin/Cummings Publishing Company, Inc., 1993.
- [3] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.
- [4] H. Gunadhi and A. Segev. Efficient indexing methods for temporal relations. *IEEE Transactions on Knowledge and Data Engineering*, 5(3), 1993.
- [5] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*, June 1984.
- [6] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD*, 1989.
- [7] P. Muth, P. O'Neil, A. Pick, and G. Weikum. Design, implementation, and performance of the LHAM log-structured history data access method. In *Proceedings of the 24th VLDB Conference*, 1998.
- [8] K. Nørnvåg. The Persistent Cache: Improving OID indexing in temporal object-oriented database systems. In *Proceedings of the 25th VLDB Conference*, 1999.
- [9] K. Nørnvåg and K. Bratbergsengen. Log-only temporal object storage. In *Proceedings of the 8th International Workshop on Database and Expert Systems Applications, DEXA'97*, 1997.
- [10] K. Nørnvåg and K. Bratbergsengen. Optimizing OID indexing cost in temporal object-oriented database systems. In *Proceedings of the 5th International Conference on Foundations of Data Organization, FODO'98*, 1998.
- [11] 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.
- [12] M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 13th VLDB Conference*, 1987.
- [13] 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.