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

Kjetil Nørvåg

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. We have previously studied OIDX performance using a relatively simple index. However, this study showed that OIDX maintenance can be very costly, and is likely to become the bottleneck of such a system. The main reason for this, is that in a temporal ODB, the OIDX needs to be updated *every time an object is updated.* This has convinced us that a new index structure, particularly suitable to TODB requirements, is necessary. In this paper, we describe an OIDX for TODBs, which we call *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. We also introduce the concept of subindex caching, which reduces the index access costs and increases the storage utilization.

1 Introduction

In a temporal object database system (TODB), object updates do not make previous versions unaccessible. 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 system, 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 identified 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 of the objects in the database. To keep these accesses as efficient as possible, and benefit from object clustering, the database is partitioned, with current objects in one partition, and the

* Revised version of IDI Technical Report 3/99, ISSN 0802-6394.

previous versions in the other partition, in the *historical database*. When an object is updated in a TODB, the previous version is first moved to the historical database, before the new version is stored in-place in the current database. The OIDX needs to be updated *every time an object is updated* (but note that as long as the modified ODs are written to the log before commit, we do not need to update the OIDX itself immediately). Even if the use of index entry caching in main memory [12] and on persistent storage [9] can be used to reduce the access cost, an efficient and flexible OIDX is still necessary.

We have in a previous paper [12] studied OIDX performance with a relatively simple, straightforward, index. The studies have shown that OIDX maintenance can be quite costly, especially when updating objects. This has convinced us that a new index structure is necessary, especially suitable to the TODB requirements. This work, which is done in the context of Vagabond Parallel Temporal ODB [10], will be described in this paper. Because Vagabond is designed to be a parallel TODB, the index structure must also support index partitioning and object declustering.

The organization of the rest of the paper is as follows. In Sect. 2 we give an overview of related work. In Sect. 3 we describe how multiversion indexing can be done efficiently in TODBs. In Sect. 4 we describe the Vagabond Temporal OIDX (VTOIDX) in detail, including an overview of the physical data organization and algorithms. In Sect. 5 we give a short overview of Vagabond, and an overall description of the parallel OIDX, and finally, in Sect. 6, we conclude the paper.

2 Related Work

OID indexing alternatives in traditional, non-temporal, ODBs has been studied by Eickler et al. [2]. We have in a previous paper developed a cost model of OID index lookup cost in non-temporal ODBs, and studied how memory can be best utilized in buffering of OIDX pages and index entries [11]. This study was extended to temporal ODBs [12]. 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 [7], R-tree [6], or LHAM [8]. 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 [16].

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. The ODs also contain other administrative information, including the timestamp of the object version. In Vagabond, we use one object descriptor (OD) *for each version of an object*. 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 discuss different multiversioning alternatives, starting with a look on some characteristics of OIDs and OID search.

3.1 Characteristics of OIDs and OID Search

When considering appropriate index structures and operations on these indexes, it is important to keep in mind 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).
- 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, the 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 object 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, as will be explained later in this paper, 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 temporal ODB, 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 what 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.

3.2 One Index Structure

If only one index is used, we have the choice of using a composite index, which is an extension of the tree based indexes used in non-temporal 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 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 objects as well as retrieval of ODs for objects created during a particular time interval can be done efficiently.

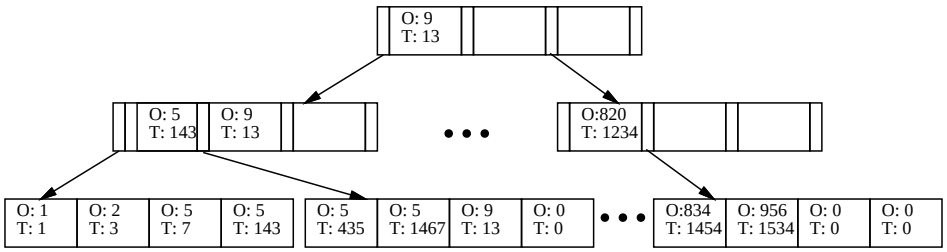

Fig. 1. One-index structure using the concatenation of OID and commit time, $OID||TIME$, as the index key.

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. Access to the ODs of current versions of objects will not be efficient. 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 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. 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 [16], based on the Texas object store [14]. 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 have 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 [7], R-tree [6], or LHAM [8], 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.

3.3 Index with Version Linking

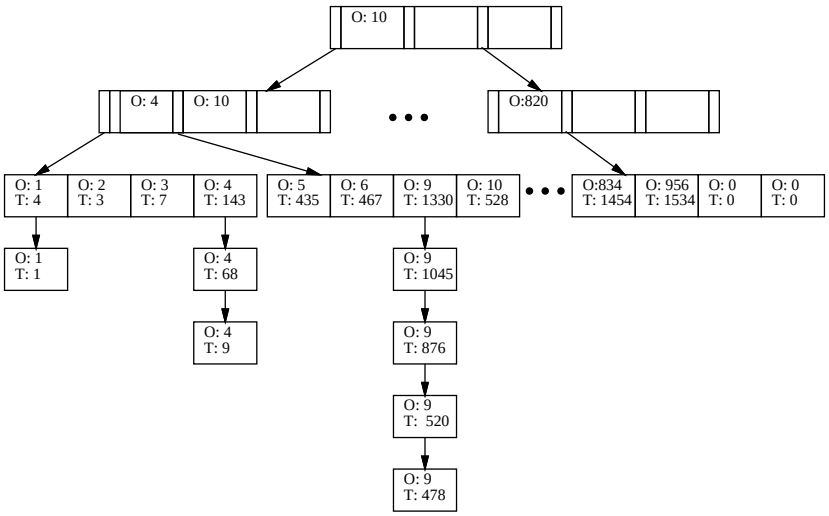

Fig. 2. One-index structure, with version linking.

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 have a list of ODs of the historical versions, as illustrated in Figure 2, 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 [15], where a link exists from one version of a tuple to the next.

A linked list approach, whether it is the ODs or the objects that are linked, has some serious disadvantages. For all operations on temporal versions, the list must be traversed, resulting in extra disk accesses.

3.4 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 [5], illustrated in Figure 3. 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.

3.5 Separate Indexes for Current and Historical Versions

In many application areas, most of the queries will be against the current data. To make read accesses to current version as efficient as possible, one index for ODs of current versions of

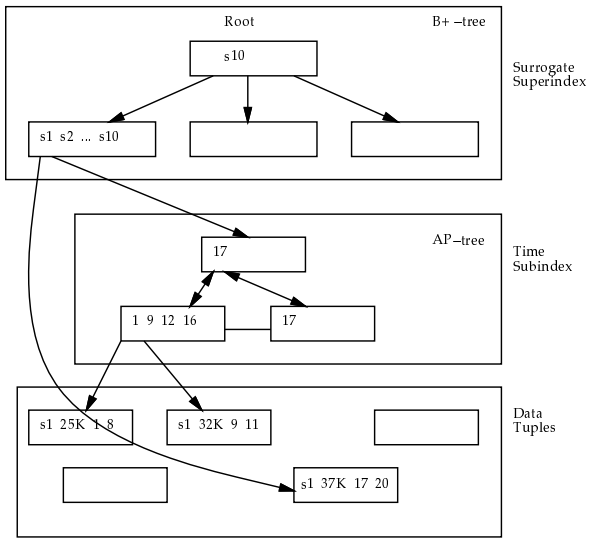

Fig. 3. Nested ST indexing [5].

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 a 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 OID Index

Our main goals in the design of the Vagabond Temporal OID Index (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.
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 use of physical containers, before we describe the indexing approach in more detail, first the physical data organization, and then the operations on the index.

4.1 Improving Index Clustering: 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, or relations). Which container to put an object in, 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 ODs will reduce the cost of index accesses.

All objects in a database are member 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.[1] Scan and query against collections can then be done efficiently. It is important to note that in other ODBs, where an 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.

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 group where the object was created. This is only used in a distributed system, see Sect. 5.
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.

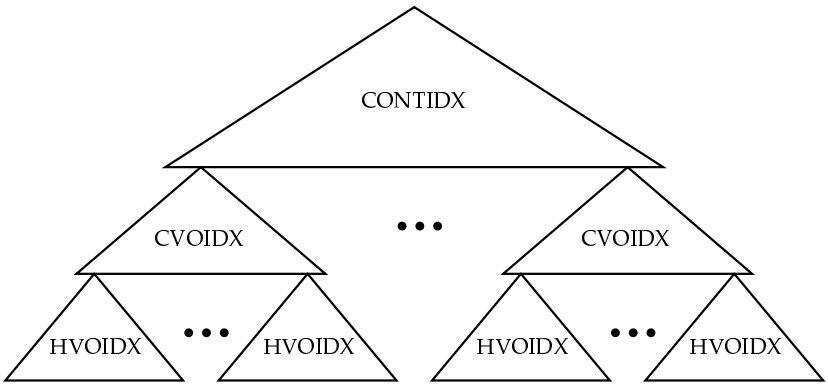

Fig. 4. The Vagabond temporal OID index.

4.2 VTOIDX Physical Data Organization

Based on the analysis of the different OID index alternatives, we have designed the Vagabond temporal OID index (VTOIDX). The VTOIDX is an hierarchy of multi-way tree indexes, with three levels, as illustrated in Figure 4:

1. Container index (CONTIDX), which indexes the physical containers in a database.
2. Current version OID index (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 outweights 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 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 OID index, one for each container. The entries in internal nodes as well as leaf nodes are (*CONTID*,pointer) tuples.

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

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 OID Index. 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 Sect. 3.4), and the CVOIDX itself is similar to the surrogate superindex in the ST index.

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

The leaf nodes of the CVOIDX contain the ODs of the current version of the objects in the container. Similar to the entries in the internal nodes of the CVOIDX, we do not store the *SGID* and *CONTID* part of the OID in the OD, 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 OID Index. For each leaf node in the CVOIDX, there is a separate HVOIDX subindex tree, with OD of the non-current versions of objects that resides or have resided in the actual leaf node. The HVOIDX is similar to the subindex in the ST index (see Sect. 3.4), 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 is index with 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 in the 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^+-tree insert algorithm, contents in a split node is 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 from the Monotonic B^+-tree [3]. 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, it will be removed. This is commonly used in B^+-tree-implementations, because 1) merging is costly, 2) in general, there is a certain risk that a split might happen again in the near future, and 3) in practice, this strategy does not result in low space utilization [4]. In the CONTIDX and CVOIDX we know that *CONTID*s 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 relies on background reorganization of the indexes to compact index page with low space utilization.

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

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

4.3 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 *CONTID*s, 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 other objects. 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^+-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, 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^+ 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 are inserted into the HVOIDX subindex (the tombstone OD is an OD where physical location is NULL, and the timestamp is the commit time of the transaction that deleted it).

As 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, it will 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. An old version of an object can not be updated or deleted. However, old versions can be *vacuumed*.[2] In this case, ODs residing in the HVOIDX will be deleted. This is done according to the standard B^+ 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 resides 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.

[2] 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).

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

Search for Current Version of all Objects in a Container. This is the traditional scan operation. In the VTOIDX, this is done in the same way as in a traditional B^+-tree, by returning 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.

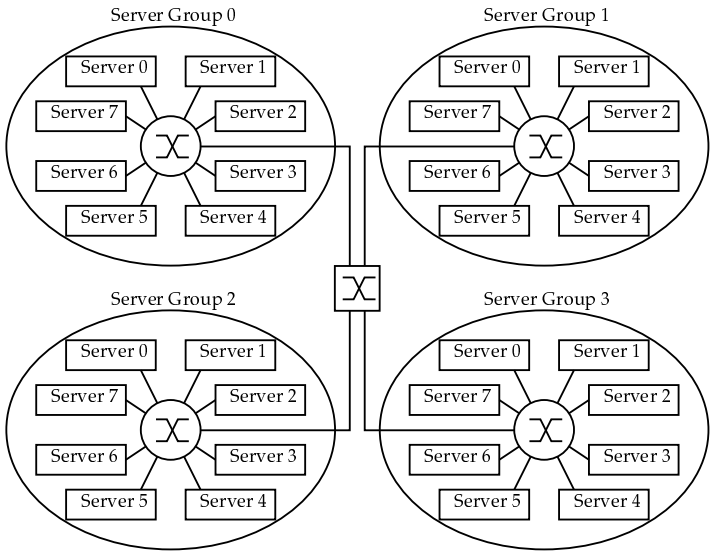

Fig. 5. Vagabond system architecture, with server groups and servers.

4.4 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^+-tree operations more efficient and improve concurrency in the VTOIDX.

4.5 Large Objects.

All objects smaller than a certain threshold, which can be different for different object classes, are written as one contiguous object on disk, while objects larger than this threshold are segmented into *subobjects*, and a *subobject index* is maintained. The subobject index we use is a variant of the EXODUS large storage objects [1].

5 Declustering of Objects and the OIDX

The Vagabond ODB is a system designed for high performance, and one strategy to achieve this, is to base the design on the use parallel servers. Data is declustered over a set of servers, which we call a *server group*. The servers in a server group can cooperate on the same task. In this way, it is possible to get a data bandwidth close to the aggregate bandwidth of the cooperating servers. To benefit from the use of a parallel server groups, it is supposed that the servers in one server group are connected by some kind of high speed communication. In many organizations it is also desirable to have the data in a *distributed system*, and the demand for support of distributed databases is increasing. To satisfy this, we use a hybrid solution: *a distributed system, with server groups* (Fig. 5). In this way, objects are clustered on server groups based on locality as is common in traditional distributed ODBs, but one server group can contain more than one computer (a kind of "super server"). Objects to be stored

on a server group are declustered on the servers in the group according to some declustering strategy.

The OIDX and the objects are declustered so that one node only indexes the objects residing on that node. Several object declustering strategies can be employed, hash partitioning based on OID is one of them. Different declustering strategies can be employed on different collections. If a collection is re-declustered, the only operations that have to be done, is to move the objects in the collection according to the new declustering strategy, and update the index at each node. No update of references in other objects is necessary, and no forwarding will be used. This is very important, relatively low cost at re-declustering time, and no additional cost later!

6 Conclusions

OID indexing in TODBs pose great challenges. Because of the update costs, it can easy become the bottleneck in such systems. Previous studies of OID indexing in TODBs have shown that achieving acceptable performance can be difficult if not most of the OIDX fits in main memory. In this paper, we have described an index structure, the VTOIDX, that we expect to 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. In a system where data is not physically deleted, either vacuuming or tertiary storage migration will be necessary.

References

1. M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the 12th VLDB Conference, Kyoto, Japan, August 1986*, 1986.
2. A. Eickler, C. A. Gerlhof, and D. Kossmann. Performance evaluation of OID mapping techniques. In *Proceedings of the 21st VLDB Conference*, 1995.
3. R. Elmasri, G. T. J. Wuu, and V. Kouramajian. The time index and the monotonic B^+-tree. In A. U. Tansel, J. Clifford, S. K. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal databases: theory, design and implementation*. The Benjamin/Cummings Publishing Company, Inc., 1993.
4. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.
5. H. Gunadhi and A. Segev. Efficient indexing methods for temporal relations. *IEEE Transactions on Knowledge and Data Engineering*, 5(3), 1993.
6. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*, June 1984.
7. D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD*, 1989.
8. 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.
9. K. Nørvåg. The Persistent Cache: Improving OID indexing in temporal object-oriented database systems. In *Proceedings of the 25th VLDB Conference*, 1999.
10. K. Nørvåg and K. Bratbergsengen. Log-only temporal object storage. In *Proceedings of the 8th International Workshop on Database and Expert Systems Applications, DEXA'97*, 1997.
11. K. Nørvåg and K. Bratbergsengen. An analytical study of object identifier indexing. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications, DEXA'98*, 1998.
12. K. Nørvåg and K. Bratbergsengen. Optimizing OID indexing cost in temporal object-oriented database systems. In *Proceedings of the 5th International Conference on Foundations of Data Organization, FODO'98*, 1998.

13. B. Salzberg and V. J. Tsotras. A comparison of access methods for time evolving data. Technical Report NU-CCS-94-21, Northeastern University, 1994.
14. V. Singhal, S. Kakkad, and P. Wilson. Texas: An efficient, portable persistent store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, 1992.
15. M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 13th VLDB Conference*, 1987.
16. 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.