A Performance Evaluation of Online Warehouse Update Algorithms*

Alexandros Labrinidis
labrinid@cs.umd.edu

Nick Roussopoulos†
nick@cs.umd.edu

Department of Computer Science and Institute for Systems Research,
University of Maryland, College Park, MD 20742

Abstract

Data warehouse maintenance algorithms usually work off-line, making the warehouse unavailable to users. However, since most organizations require continuous operation, we need be able to perform the updates online, concurrently with user queries. To guarantee that user queries access a consistent view of the warehouse, online update algorithms introduce redundancy in order to store multiple versions of the data objects that are being changed. In this paper, we present an online warehouse update algorithm, that stores multiple versions of data as separate rows (*vertical redundancy*). We compare our algorithm to another online algorithm that stores multiple versions within each tuple by extending the table schema (*horizontal redundancy*). We have implemented both algorithms on top of an Informix Dynamic Server and measured their performance under varying workloads, focusing on their impact on query response times. Our experiments show that, except for a limited number of cases, vertical redundancy is a better choice, with respect to storage, implementation overhead, and query performance.

†Also with the Institute for Advanced Computer Studies, University of Maryland.

*Prepared through collaborative participation in the Advanced Telecommunications/Information Distribution Research Program (ATIRP) Consortium sponsored by the U.S. Army Research Laboratory under the Federated Laboratory Program, Cooperative Agreement DAAL01-96-2-0002. Also supported in part by DOD Lucite Contract CG9815.

1 Introduction

Data warehouses contain replicated data from several external sources, collected to answer decision support queries. The degree of replication is further extended by introducing other derived data to facilitate query processing and maintenance. The derived data include all kinds of indices, materialized views, summary tables, multidimensional aggregate views such as the data cube, and so on. We refer to all these with the most general term "materialized views" ([Rou98]).

When data at the external sources change, updates are sent to the warehouse, which has to perform a *refresh* operation. Except for updating the base data tables, materialized views also need to be updated in order for the warehouse to reach a fully consistent state. Executing the refresh operation poses a serious trade-off. On the one hand, the sheer volume of data stored in warehouses, makes the refresh operation a costly procedure. On the other hand, in order to avoid having stale data, the warehouse needs to be refreshed frequently. In current business practices, the warehouse is taken *off-line* for the duration of the refresh operation. During refresh, the warehouse is down and no queries are allowed to run (since they would access inconsistent data) and because of that, the refresh operation is usually scheduled overnight. However the new world order of globalization in operation shrinks or completely eliminates the overnight down-time window ([DDWJR98]), since it is always daytime in some part of the world.

One possible solution is to try to *minimize down-time* and thus make the effects of the warehouse being off-line as little as possible ([CGL$^+$96]). Another, even better, solution is to eliminate down-time altogether, by using an *online* update algorithm, and thus accommodate the continuous operation requirement of organizations ([B$^+$98]). Such an online algorithm would allow the user queries to run concurrently with the refresh process at the warehouse, assuming that the queries are protected from accessing inconsistent data.

As a first approach towards an online algorithm, one could simply keep two separate copies of the entire database: one only for the readers and one only for the maintenance operations. In this setup, the refresh operation does not interfere with read-only queries. Except for the huge storage overhead however, there is a big cost to pay, when the updated database is copied over the read-only database, which renders this approach impractical.

For a second approach towards an online algorithm, one could rely on traditional database technology and "enclose" the refresh operation in one long transaction, which would guarantee[1] that all the warehouse queries will be able to access the changes to the warehouse only after the refresh is complete. This long update transaction should include both the updates to the base table data and the incremental updates to all the affected materialized views, in order to bring the warehouse into a fully consistent state. Obviously, such a transaction would be prohibitively long and hence is not a plausible solution.

A better idea is to keep *multiple versions* of the data objects stored in the warehouse (i.e. in-

[1] If running with at least isolation level 2 ([BBG$^+$95]).

troduce *redundancy*), and with some bookkeeping, always be able to present a fully consistent version of the warehouse to the queries while the warehouse is being updated. Multiversioning has been used extensively to provide concurrency control and recovery in (distributed) database systems ([Ree83, SR81, CFL$^+$82, BS83, BG83, AS93, SA93, MWYC96, JMR97, LST97]). Specialized multiversion access structures have also been proposed ([LS89, LS90, dBS96, BC97, VV97, MOPW98])

In the context of OLTP systems, long read-only queries can be allowed to access an older, fully consistent version of the database, while the update transactions are operating on a different version of the database ([BC92b, BC92a]). This approach is known as *transient versioning* ([MPL92]), since the multiple versions of data objects are not kept forever, in contrast to *historic databases*.

The main difference between OLTP systems and data warehousing systems, is that the updater process in data warehouses is usually *only one* (but long), compared to the many short update transactions that OLTP systems usually have. This means that in the data warehouse case there are no write conflicts, but there still is a lot of data contention between the updater process and user queries (which are typically long & complex).

The absence of write conflicts, allows for simpler online update algorithms that use multiversioning, but don't require locking. Quass & Widom presented such an algorithm, *2VNL*, in [QW97], which extends each tuple in order to store the "before" values (*horizontal redundancy*). We present another alternative where multiple versions are stored as multiple tuples (*vertical redundancy*). For both online update algorithms, the update process is expected to run in the background, and from time to time "release" new, fully consistent versions of the warehouse for queries to access. Old versions of objects can be deleted later, after making sure that no user query accesses them.

In the next section we present the details of our online update algorithm. In section 3 we briefly describe horizontal redundancy and compare it to vertical redundancy. In section 4 we present the results of our experiments, followed by our conclusions.

2 Vertical Redundancy: *MVNL*

One way to support online warehouse updates is by *vertical redundancy*: multiple versions of each tuple are stored as multiple rows in the table. By adding control information to each record, queries can always access a transaction consistent view of the warehouse, and the refresh process does not delay readers as it does not have to acquire any locks.

Of course, supporting online warehouse updates does not come for free. Except for the minor schema changes, there must be a distinction between *logical* and *physical* update operations. For example a tuple deletion cannot result in the tuple being physically deleted right away, as it might be used by the readers. Instead, it has to be just marked as deleted and be actually deleted at a later time (garbage collection). Similarly, updates on data cannot be done in-place, but instead, care must be taken so that the old values remain accessible to the readers.

In the following paragraphs we present the details of our algorithm, *MVNL*, which stands for *Multi-Version No Locking* and is named after the classic MV2PL algorithms on which it is based. First we give a short introduction on how versions work in the system, followed by a list of modifications required to support the algorithm. Then we outline our approach on garbage collection. Last we describe a mechanism to track version numbers among the updater, the query and the garbage collection processes.

2.1 Multiversioning

MVNL supports multiple versions by using Time Travel ([Sto87]). Each row has two extra attributes, T_{min}, the *insertion timestamp*, and T_{max}, the *deletion timestamp*. The insertion timestamp gets assigned when the tuple is first inserted into the database, whereas the deletion timestamp gets assigned when the tuple is marked as deleted. These timestamps are used by queries to filter out rows that are "younger" than the queries and are not supposed to be "visible".

There are three timestamp variables in the system:

- $\boldsymbol{T_{maint}}$ is a private version number[2] counter that is used by the maintenance process. It is initialized at 1, and is incremented by 1 each time a warehouse refresh operation has been completed. It corresponds to the version of the database that the maintenance process is currently updating. Tuples that are inserted get $T_{min} = T_{maint}$ and $T_{max} = \infty$, whereas tuples that are marked deleted get $T_{max} = T_{maint}$.
- $\boldsymbol{T_{safe}}$ is a global version counter, that the maintenance process updates every time a refresh operation is completed. It corresponds to the maximum (most recent) version in the database that is consistent.

[2]We use the terms *timestamp* and *version number* interchangeably.

- $\boldsymbol{T_{query}}$ is a private variable, which is initialized with the current value of T_{safe}, at the start of each query process. It corresponds to the version of the database that the query is allowed to access. User queries are only allowed to access tuples that were created sometime in the past, and have not yet been marked as deleted. In timestamp arithmetic, the visible tuples should have $T_{min} <= T_{query}$ and $T_{query} < T_{max}$ (remember that if a tuple is not deleted it has $T_{max} = \infty$).

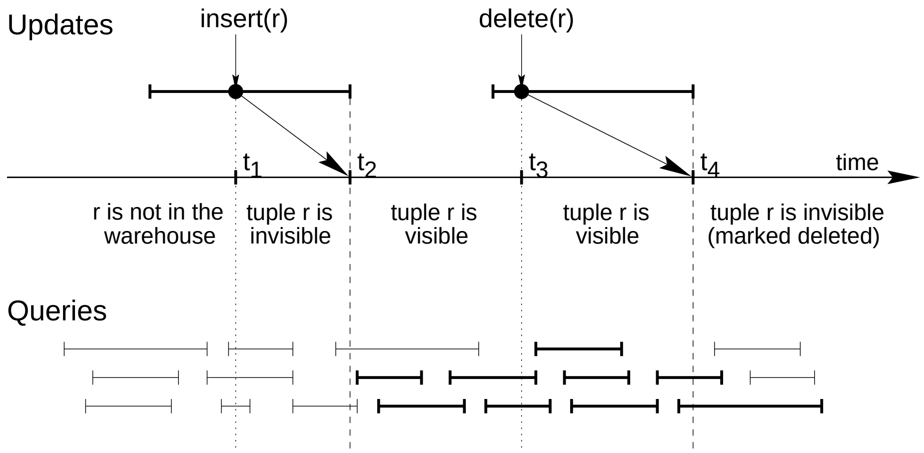


Figure 1: Life cycle of a tuple

To illustrate the use of timestamps, we follow the life-cycle of a tuple r (see figure 1). At some point in time, t_1, tuple r will be inserted into the database by the update process. However, the tuple will not be visible to user queries until the update process finishes with the entire batch of updates, at t_2. At that point, T_{safe} will be updated and the new version will be released. Queries starting *after* t_2 will be "allowed" to access tuple r. The deletion of tuple r works similarly. Although the tuple gets marked as deleted (at point t_3), this fact will not be visible to user queries until the update process finishes the entire batch of updates and releases the new version at t_4. Queries starting after t_4 will "see" that tuple r has been marked deleted.

2.2 Modifying the relation schema

In order to support multiversioning, we must modify the schema only for the relations that might get updates. The required changes are straightforward after the discussion in the previous section. We simply need to add two integer attributes, T_{min}, to store the insertion timestamp, and, T_{max}, to store the deletion timestamp. In other words, if the pre-*MVNL* schema for a relation was $(a_1, a_2, \ldots, a_k)$, where a_i is an attribute and k is the total number of attributes, it will have to be modified into:

$$(T_{min}, T_{max}, a_1, a_2, \ldots, a_k)$$

We can calculate the storage overhead for keeping the extra versions of data in multiple rows. Let us assume that initially, relation R has k attributes $\{a_1, a_2, \ldots, a_k\}$, and N rows, each of size

$S_{\text{base}} = \sum_{1\ldots k} \texttt{sizeof}(a_i)$. Also assume that there were $N_{\mathcal{I}}$ new rows inserted to the relation, $N_{\mathcal{D}}$ rows deleted, and $N_{\mathcal{M}}$ rows modified. If A_M is the set of all attributes from relation R that get modified, then let S_{mod} be the total size of the all these attributes, or $S_{\text{mod}} = \sum_{a_i \in A_M} \texttt{sizeof}(a_i)$.

If we wish to apply the updates but also keep the previous version of the data, we will need to keep the deleted tuples and also store the modifications separately. In this case, the minimum total storage requirement would be:

$$\mathit{TS}_{\text{normal}} = (N + N_{\mathcal{I}}) \times S_{\text{base}} + N_{\mathcal{M}} \times S_{\text{mod}} \tag{1}$$

Under *MVNL* we store extra control information together with all the database tuples, and we also keep the previous versions of the data as separate tuples. The storage requirement is:

$$\mathit{TS}_{\mathit{MVNL}} = (N + N_{\mathcal{I}} + N_{\mathcal{M}}) \times (S_{\text{base}} + S_{\mathit{MVNL}}) \tag{2}$$

where $S_{\mathit{MVNL}} = \texttt{sizeof}(T_{min}) + \texttt{sizeof}(T_{max})$.

2.3 Modifying the updater

We assume that updates from the external data sources arrive at the data warehouse[3] asynchronously, but are delivered *in-order*[4] , i.e. updates from the same source arrive at the warehouse in the same order by which they were sent. The continuous update stream is split into *batches* of work by the updater process. In order to guarantee that these batches do not cross transaction boundaries, data sources annotate the update streams with transaction begin/end markers which are later observed when splitting the stream. Each of these batches will correspond to one version of the warehouse.

The updater process keeps a private version counter, T_{maint}, which uses to "mark" all the changes it makes to the warehouse. All update operations that belong to the same batch get marked with the same version number. When the batch is complete, the value of T_{maint} is made public by assigning $T_{safe} = T_{maint}$ (in effect "releasing" that version) and is also incremented by 1. This protocol ensures that either all of the updates in a batch are visible to the queries or none of them.

In order for the updater process to support multiversioning, it must make the distinction between *logical* and *physical* update operations. Logical operations are those suggested by the data sources, and physical operations are the ones that will actually be executed because of multiversioning through *MVNL*. For example, a deletion, cannot be executed right away, as there might be queries still accessing the version of the warehouse the deleted tuple belongs to.

In the following paragraphs, we give details on the mapping between logical and physical update operations, which is dictated by *MVNL*. In our examples, relation R, has k attributes: $(a_1, a_2, \ldots, a_k)$, of which a_p is the primary key, and v_i is the value that corresponds to attribute a_i.

[3]We use the terms *database* and *warehouse* interchangeably.

[4]If ordered delivery is not guaranteed, it can be implemented with a simple sequencing scheme at the data sources and a slight modification of the updater.

Logical operation	Physical operation(s)
`insert values (` $v_1, v_2, \ldots, v_k$ `);`	`insert values (` $T_{maint}, \infty, v_1, v_2, \ldots, v_k$ `);`
`delete from R where` $a_p = v_p$ `;`	`update R set Tmax =` T_{maint} `where` $a_p = v_p$ `;`
`update R set * = (` $v_1, v_2, \ldots, v_k$ `)` `where` $a_p = v_p$ `;`	`update R set Tmax =` T_{maint} `where` $a_p = v_p$ `;` `insert values (` $T_{maint}, \infty, v_1, v_2, \ldots, v_k$ `);`

Table 1: Mapping of logical to physical update operations

Insertions Physical insertions are almost identical to logical insertions. To support *MVNL* we only need to store the version information at each tuple. So, a logical *insert*($v_1, v_2, \ldots, v_k$) is translated into a physical *insert*($T_{maint}, \infty, v_1, v_2, \ldots, v_k$).

Deletions As explained earlier, logical deletions cannot be translated directly to physical deletions since other queries should be able to have access to the tuple that is to be deleted. Therefore, instead of deleting such tuples, we simply mark them as deleted. So, a logical *delete*(v_p), is translated into a physical *update*($T_{max} \leftarrow T_{maint}$, where $a_p = v_p$). At a later time, when no reader needs to access the tuple, it gets physically deleted by garbage collection.

Updates The handling of updates in *MVNL* adheres to the *copy-on-write* principle. Since old versions of data are possibly needed by readers, the updater cannot perform tuple modifications in place. Instead, all logical update operations are mapped into a pair of logical deletion and logical insertion operation, which are then translated, as described earlier, into a physical update and a physical insert operation. So, a logical *update*($a_i = v_i$, where $a_p = v_p$), is mapped into a logical *delete*(v_p) and a logical *insert*($v_1, v_2, \ldots, v_k$). These two are then translated into a physical *update*($T_{max} \leftarrow T_{maint}$, where $a_p = v_p$) and a physical *insert*($v_1, v_2, \ldots, v_k$).

Example 1 *Let us consider the* `order` *table from the TPC-D benchmark. Assume that an updater process (with* $T_{maint} = 100$*) wants to insert into* `order` *the information on the order with* `orderkey` *= 12345. After the insertion, the* `order` *table will look like this:*

tmin	*tmax*	*orderkey*	*custkey*	*orderstatus*	...	*comment*
100	∞	*12345*	...	*P*	...	*Special kids wrapping needed*

Now suppose that at a later time, there needs to be an update to this entry, for example a change in the orderstatus (from Pending to Completed). If the updater process that performs this change has $T_{maint} = 103$*, then the* `order` *table will look like this:*

tmin	*tmax*	*orderkey*	*custkey*	*orderstatus*	...	*comment*
100	*103*	*12345*	...	*P*	...	*Special kids wrapping needed*
103	∞	*12345*	...	*C*	...	*Special kids wrapping needed*

Finally, suppose that after a long time, this entry needs to be deleted. If the updater process that performs this change has $T_{maint} = 202$, then the `order` *table will look like this:*

tmin	*tmax*	*orderkey*	*custkey*	*orderstatus*	...	*comment*
100	*103*	*12345*	...	*P*	...	*Special kids wrapping needed*
103	*202*	*12345*	...	*C*	...	*Special kids wrapping needed*

Example 1 brings up a few points worth mentioning. First of all it is clear that the readers need to be modified in order to "distinguish" the right version of the data, since multiple versions are kept in the warehouse. We describe the necessary modifications for the readers in the next section. Secondly, it is also clear that having all those tuples hanging around after they are logically deleted will pose some read overhead. Periodically, the garbage collection will clean tuples that are marked as deleted, but there is a trade-off between read overhead and the one imposed by garbage collection. Finally, one might notice that in the update operation, we had to duplicate the entire row, despite the fact that only one attribute was modified. In the general case we don't know in advance which tuples can be modified, but if we can restrict the set of the updateable attributes, then we might use a different approach. Section 3.1 briefly describes a solution presented in [QW97] which is based on that observation.

2.4 Modifying the readers

Each reader process, upon startup, initializes a private variable, T_{query}, to the current "released", fully consistent version of the warehouse (= T_{safe}). T_{query} is used as a guide to filter out tuples that are supposed to be invisible to the reader. More specifically, the reader process should only access tuples that have:

$$T_{min} <= T_{query} \quad \text{and} \quad T_{query} < T_{max} \tag{3}$$

The first part of the expression simply prohibits accessing any newly inserted tuples, whereas the second part guarantees that tuples marked as deleted in the past will not be visible to the query. All user queries have to be rewritten using the query modification technique ([Sto75]) to include this constraint in their `where` clause.

A positive side-effect of this approach is that, in effect, it guarantees a Repeatable Read isolation level ([BBG+95]) for the readers, since the value of T_{query} stays the same for the duration of the user query.

2.5 Garbage Collection

Periodically we will need to physically delete the tuples that have been marked as deleted, but do not belong to a version that is currently being used by a reader. Although this garbage collection procedure reduces read overhead, it is not necessary for correctness. Readers in *MVNL* will always see a consistent view of the database no matter how many old versions are kept. The reason behind garbage collection is *performance*. By removing unused tuples we reduce the storage overhead imposed on the warehouse by the online algorithm. This means that relation scans will be shorter and indexes will be smaller, leading to faster query responses.

Garbage collection should be executed periodically, or when server load permits it. It can run concurrently with the updater, and the readers. To perform garbage collection, we need to know the highest version number that is not being accessed by any reader, T_{kill}. The SQL to remove the *"invisible"* tuples is then:

`delete from R where Tmax <=` T_{kill}`;`

In order to find T_{kill} there has to be some minimal coordination between the updater, the readers and the garbgage collection process, which we describe in the next paragraph.

2.6 Version Management

All of the processes that we have described so far (Updater, Readers, Garbage Collector) should be able to run concurrently, with almost no interaction among them. The only points of coordination that exist are the setting of the T_{safe} and T_{kill} variables. The T_{safe} variable needs to be assigned to the value of T_{maint} after each update batch is completed, so that queries can find out the number of the latest consistent version (i.e. $T_{maint} \rightarrow T_{safe} \rightarrow T_{query}$). The T_{kill} variable holds the highest version number not being accessed by any reader.

We present one possible implementation of this coordination through a relational table, `veta` (short for VErsion TAble), but it should be fairly straightforward to implement it in main memory instead[5] . In order to store both T_{safe} and the T_{query} variables of all active readers, table `veta` has two attributes:

- *type*, which can take two values: `'U'` for Updater or `'R'` for Reader, and,
- *vn*, which stores the version number. If *type* = `'U'` then *vn* = T_{safe}, otherwise *vn* = T_{query} of one reader.

After describing the schema for `veta`, we will present in detail the steps the Updater, Reader, Garbage Collection processes need to take in order to coordinate with each other.

[5]However, a main memory implementation might not be significantly faster than the relational table one, as we expect the table to remain in main memory since it is really small.

The updater, upon completion of a batch of updates, will "release" the current version by updating the `veta` table as follows:

`insert in veta values ('U',` T_{maint}`);`
`delete from veta where type = 'U' and vn =` $T_{maint} - 1$ `;`

After this is completed, the updater will increment its private T_{maint} variable.

The reader, upon startup, will first read the current T_{safe}:

T_{safe}`= select max(vn) from veta where type = 'U';`

and after it copies it to its private T_{query} variable ($T_{query} \leftarrow T_{safe}$), it will record in `veta` the fact that it is using this version:

`insert into veta values ('R',` T_{query}`);`

The reader, upon completion, should record in `veta` the fact that it is no longer using that version (so that the garbage collector will be able to identify which versions are currently active):

`delete from veta where type = 'R' and vn =` T_{query}`;`

Finally, the garbage collection process, upon startup should determine the minimum version number currently in use and deduct one to get the maximum version that is not being used (and can be deleted):

$T_{kill} =$ `(select min(vn) from veta where type = 'R')` $- 1$

3 Vertical Redundancy vs Horizontal Redundancy

Another way to support online warehouse updates is with *horizontal redundancy*: multiple versions of data objects are stored *within* the same tuple, by extending the table schema. Control information at each record allows queries to always calculate a transaction consistent view of the warehouse, while the refresh process does not delay readers as it does not have to acquire any long-term locks.

In the following sections we briefly describe horizontal redundancy, compare it to vertical redundancy and, finally, give some details about their implementation.

3.1 Horizontal Redundancy: *2VNL*

Quass and Widom presented in [QW97] an online warehouse update algorithm, *2VNL*, that uses horizontal redundancy. The idea behind the algorithm is to extend each tuple to hold the "before" values of the attributes that change. Up to two different versions of the data are stored in the warehouse, one being used by the update process, and one being accessed by user queries, which allows user queries to run concurrently with the update process and always "see" a consistent view of the warehouse. To implement *2VNL* one has to make changes in the relation schema, in the update process and in the user queries. We outline these modifications in the following paragraphs.

Modifying the relation schema Before we augment the existing relation schema to support horizontal redundancy, we need to identify which attributes from each relation are *updateable*, i.e. might be modified by an update statement. When deciding if an attribute is updateable or not, we should always take a conservative approach: first characterize all attributes as updateable by default, and then, only if we can guarantee that for the entire life of our system there will not be an update operation on an attribute, we could exclude it from the list. This conservative approach is necessary since we won't be able to service an update operation on an attribute that has been deemed non-updateable.

Let R be a relation with k attributes: $(a_1, a_2, \ldots, a_k)$, and suppose m of these attributes are updateable: $\{a_{x_1}, a_{x_2}, \ldots, a_{x_m}\}$, where $1 \leq x_i \leq k$. The extended relation schema would then be:

$$(\textit{tupleVN},\ \textit{operation},\ a_1, a_2, \ldots, a_k,\ a_{x_1}, a_{x_2}, \ldots, a_{x_m})$$

where *tupleVN* contains the version number of the maintenance process that performed the last operation on the tuple and *operation* is the last operation performed (insert, delete or update). Clearly, the worst case would be when $m = k$, where we would have to approximately double the size of the warehouse.

We can calculate the storage requirement for keeping the before values of data by extending each tuple. Let us assume that relation R has N rows initially, each of size $S_{\text{base}} = \sum_{i=1}^{k} \texttt{sizeof}(a_i)$.

Also, let S_{mod} be the total size of all the updateable attributes, or $S_{\text{mod}} = \sum_{i=1}^{m} \texttt{sizeof}(a_{x_i})$. Finally, let us assume that there were $N_{\mathcal{I}}$ new rows inserted to the relation, $N_{\mathcal{D}}$ rows deleted, and $N_{\mathcal{M}}$ rows modified. Under *2VNL*, we need to allocate extra space to store the before-values of the updateable attributes for all tuples in the warehouse. The total storage requirement for relation R would be:

$$TS_{2VNL} = (N + N_{\mathcal{I}}) \times (S_{\text{base}} + S_{\text{mod}} + S_{2VNL}) \tag{4}$$

where $S_{2VNL} = \texttt{sizeof}(\textit{tupleVN}) + \texttt{sizeof}(\textit{operation})$.

Modifying the updater To support multiversioning through horizontal redundancy we must again make the distinction between logical and physical update operations:

- Logical insertions, are translated into physical insertions with the addition of the *tupleVN* and *operation* attributes.
- Logical deletions, are translated into physical updates, where the tuple is simply marked as deleted by properly setting the *tupleVN* and *operation* attributes.
- Logical updates, are translated into physical updates. Care is taken so that the old values of the attributes, are "copied" to the area inside each tuple allocated to store the before-values.

One other rule that applies to all kinds of update operations under *2VNL* is that, in order for the algorithm to work, we need to identify the *net effect* of operations that are encountered inside the same "batch". If for example we have an insertion of a tuple r, followed by an update on tuple r, then we must record the combination of these two operations as an insertion, with the inserted values being the ones after the update operation. This is not a problem in *MVNL*, since it can "tolerate" multiple instances of the same tuple in the warehouse, provided that only one is marked as valid.

Modifying the readers With *2VNL*, readers are able to access an old, but transaction consistent version of the warehouse, while the maintenance process works on a future, "un-released" version. For that reason, all user queries need to be modified to filter out tuples that are not supposed to be "visible" to them (by adding a few predicates to the `where` clause) and also choose the before-value on data items that are being changed (using CASE expressions from SQL 92).

Since there are only two versions kept, there is always the chance, if a user query is too long, that the version of the database the query was assigned upon startup will *expire* before the query finishes. This can be detected, but unfortunately the query will have to be restarted or it would access inconsistent data. The authors outline a solution to this problem which would require keeping more than 2 versions in the warehouse. We briefly discuss this in the next section.

Garbage Collection Performing logical deletions by marking the tuple as deleted and not physically removing it has the drawback of creating "garbage", i.e. warehouse tuples which after a point are not visible to any reader. Periodically, a garbage collection process can run in the background and remove those tuples from the database, reclaiming that space.

3.2 Horizontal Redundancy: *n-VNL*

With *2VNL*, reader sessions will "expire" if they span more than one maintenance transaction, and would have to be restarted. The solution to this problem is to extend the algorithm to support n versions, and thus handle the case of a reader overlapping with at most $(n-1)$ maintenance transactions. The resulting algorithm, *n-VNL*, was presented in [QW97] and is able to make n versions of the warehouse available at the same time.

The modifications to the relation schema are similar to those of *2VNL*, but instead of allocating space for one more extra version, we need to allocate space for $n-1$ versions in every tuple. So, if R is a relation with k attributes: $(a_1, a_2, \ldots, a_k)$, and m of these attributes are updateable: $\{a_{x_1}, a_{x_2}, \ldots, a_{x_m}\}$, where $1 \leq x_i \leq k$, the schema for *n-VNL* will be:

$$(\textit{tupleVN}_1, \ldots, \textit{tupleVN}_{n-1},\ \textit{op}_1, \ldots, \textit{op}_{n-1},\ a_1, a_2, \ldots, a_k,\ \underbrace{a^1_{x_1}, \ldots, a^1_{x_m}}_{2^{nd}\text{ version}},\ \ldots,\ \underbrace{a^{n-1}_{x_1}, \ldots, a^{n-1}_{x_m}}_{n^{th}\text{ version}})$$

In the worst case, where $m = k$, the size of the warehouse will grow approximately n-fold. In the general case, if relation R had N rows initially, and there were $N_{\mathcal{I}}$ new rows inserted, $N_{\mathcal{D}}$ rows deleted, and $N_{\mathcal{M}}$ rows modified, the minimum total storage requirement for R would be:

$$\textit{TS}_{\textit{n-VNL}} = (N + N_{\mathcal{I}}) \times (S_{\text{base}} + (n-1) \times (S_{\text{mod}} + S_{\textit{2VNL}})) \tag{5}$$

where $S_{\text{base}} = \sum_{i=1}^{k} \texttt{sizeof}(a_i)$, $S_{\text{mod}} = \sum_{i=1}^{m} \texttt{sizeof}(a_{x_i})$ and
$S_{\textit{2VNL}} = \texttt{sizeof}(\textit{tupleVN}) + \texttt{sizeof}(\textit{operation})$.

The updater under *n-VNL* would have to translate logical update operations into physical operations exactly like *2VNL*. The only difference is that for each new version that we have to store, we need to first "push back" the data for the previous versions, thus eliminating the n^{th} version. This shifting will cause significant overhead making the choice of n a very important design decision.

Readers would also need to be modified in a manner similar to the 2-version algorithm. However, the predicates and the CASE expressions used to provide user queries with a consistent view of the warehouse are expected to be noticeably more complex.

3.3 Comparison

Although *MVNL* and *2VNL* are both based on *multiversioning*, they have a lot of differences. We explore the most important of these differences in the next paragraphs.

Concurrent execution of readers & updater The major drawback of off-line update algorithms is that user queries cannot run while the warehouse is being updated. No user query will be allowed to start during the refresh operation and any query that ends inside the update window will have to be aborted (Fig. 2).

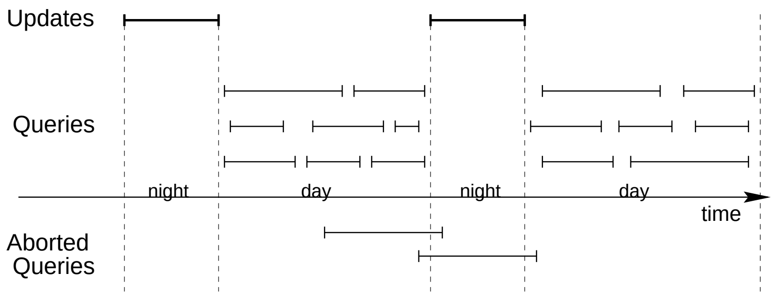


Figure 2: No Concurrent Execution: Off-line algorithms

Online algorithms on the other hand, allow for concurrent execution of the maintenance transaction and user queries by introducing redundancy. *2VNL*, which employs horizontal redundancy, stores up to two versions of warehouse data, which allows user queries to run while the warehouse is being updated. However, user queries can overlap with at most one maintenance transaction (Fig. 3). When a reader spans more than two maintenance transactions, its session will *expire* and will have to be restarted. A solution to this is to increase the amount of redundancy, by storing n versions of data (where n is specified) and use n*-VNL*. However, the storage cost of this solution is prohibitively high, as, in the worst case, the warehouse can grow n-fold in size.

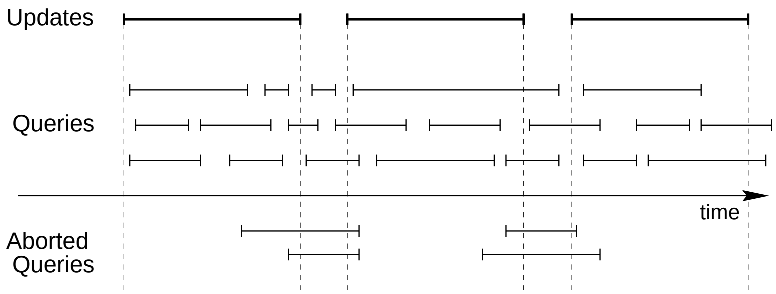


Figure 3: Concurrent Execution: *2VNL*

MVNL, which employs vertically redundancy, also enables user queries to run while the warehouse is being updated. However, unlike *2VNL*, there is no limit to the number of maintenance transactions a query can overlap with during its execution (Fig. 4). Queries can be arbitrary long, warehouse update "transactions" can be arbitrary short and they would still be allowed to run concurrently.

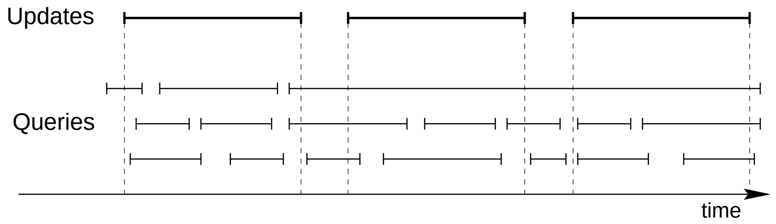


Figure 4: Concurrent Execution: *MVNL*

Storage overhead We have already calculated the minimum storage requirement for all online algorithms, but to be able to compare them we calculate the net storage overhead for each one. Recall that TS_{normal} is the minimum storage required to keep the old versions of data after applying the updates (given in Eq. 1), TS_{MVNL} is the storage requirement for *MVNL* (given in Eq. 2), TS_{2VNL} is the requirement for *2VNL* (given in Eq. 4), and $TS_{n\text{-}VNL}$ is the storage requirement for *n-VNL* (given in Eq. 5).

We calculate the storage overheads for *MVNL*, *2VNL* and *n-VNL* respectively:

$$\begin{aligned} O_{MVNL} &= TS_{MVNL} - TS_{\text{normal}} = N_{\mathcal{M}} \times (S_{\text{base}} - S_{\text{mod}}) + (N + N_{\mathcal{I}} + N_{\mathcal{M}}) \times S_{MVNL} && (6) \\ O_{2VNL} &= TS_{2VNL} - TS_{\text{normal}} = (N + N_{\mathcal{I}} - N_{\mathcal{M}}) \times S_{\text{mod}} + (N + N_{\mathcal{I}}) \times S_{2VNL} && (7) \\ O_{n\text{-}VNL} &= TS_{n\text{-}VNL} - TS_{\text{normal}} = ((n-1) \times (N + N_{\mathcal{I}}) - N_{\mathcal{M}}) \times S_{\text{mod}} \\ &\quad + (n-1) \times (N + N_{\mathcal{I}}) \times S_{2VNL} && (8) \end{aligned}$$

Clearly, the storage overhead for the n-version horizontal redundancy algorithm, *n-VNL*, is prohibitively high (even for small n since the entire warehouse population needs to be replicated).

To compare the storage overhead for vertical redundancy with the one for horizontal redundancy, we solve $O_{2VNL} > O_{MVNL}$. Assuming that $S_{MVNL} \simeq S_{2VNL}$ [6] we get:

$$O_{2VNL} > O_{MVNL} \Leftrightarrow \frac{S_{\text{mod}}}{S_{\text{base}}} > (1 + c) \times \frac{N_{\mathcal{M}}}{N + N_{\mathcal{I}}} \quad (9)$$

where S_{mod} is the size of the updateable attributes, S_{base} is the tuple size of the relation originally, $c = \frac{S_{MVNL}}{S_{\text{base}}} = \frac{S_{2VNL}}{S_{\text{base}}}$, $c \ll 1$, N is the number of rows in the relation initially, $N_{\mathcal{M}}$ is the number of rows modified by the set of updates and $N_{\mathcal{I}}$ is the number of rows inserted.

[6] In our implementation we have $S_{MVNL} = S_{2VNL} = 2 \times$ `sizeof(INTEGER)`.

We plot Eq. 9 in Fig. 5, for $c = 5\%$ and for $N_{\mathcal{M}}$ up to 10% of $(N + N_{\mathcal{I}})$. Note that in typical warehouses, the percentage of modifications is expected to be much lower (almost close to 0%) since we mostly have insertions and deletions.

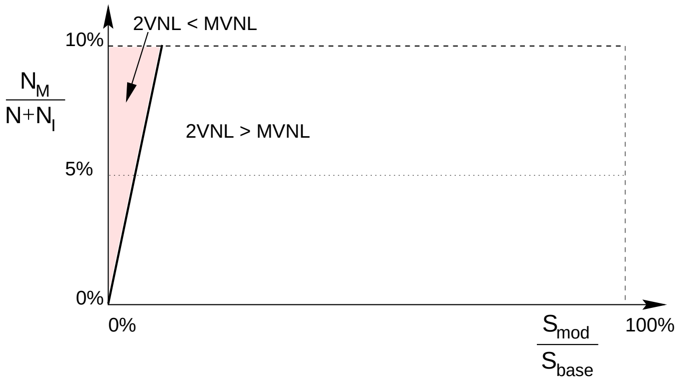


Figure 5: Storage Overhead: *MVNL* vs *2VNL*

Fig. 5 illustrates which cases favor horizontal redundancy and which favor vertical redundancy. Horizontal redundancy algorithms extend each tuple to hold the modified values, so, in effect they "assume" that the amount of modifications is going to be comparable to the size of the database. As expected, they have low storage overhead only in cases where the percentage of modifications is really high. On the other hand, vertical redundancy algorithms copy an entire tuple to hold the modifications, so, they have very low overhead when the modification is a significant portion of a tuple. Since both algorithms handle insertions of new tuples and deletions of existing tuples similarly, modifications are the operations that determine which of the two schemes is best. Overall, we can see that in typical warehouse configurations, vertical redundancy should be a more economical solution in terms of storage overhead, compared to horizontal redundancy.

Implementation complexity Implementing *MVNL* is relatively easy. The changes to be made on the relation schema are trivial (adding two integer attributes). The updater process needs to conform with the mapping of logical to physical update operations (Tab. 1), and the read-only queries have to be modified to include an extra visibility predicate.

Implementing *2VNL* is a more complex task. In order not to duplicate the entire warehouse, one has to have knowledge of the application domain and decide on the set of updateable attributes. The updater process will have to map logical update operations into the equivalent physical operations with the same "net effect". This forces the use of cursors, as the previous version of the tuple is required in order to decide what the next physical operation should be. Finally, read-only queries have to be modified in two ways: a) include an extra visibility predicate (similarly to *MVNL*), and, b) choose the current or before-value for every attribute in the projection list (through a CASE expression).

4 Experiments

We have implemented both online update algorithms, *MVNL* and *2VNL* as user defined functions (written in C) on an Informix Dynamic Server with Universal Data Option ver 9.12. In fact, we have implemented two variations of *2VNL*, one in which there is only one updateable attribute (*2VNL/1*) and one where all the attributes are updateable (*2VNL/k*). We used `dbgen`, the data generator from the TPC-D Benchmark ([Tra98]), to populate the warehouse for our experiments and also to generate the updates. However, we have made two extensions to the original generator:

- We *annotate* the update workload, with begin/end markers around each transaction[7]. This is required in order to be able to observe transaction boundaries when splitting the workload in batches.
- We add a small percentage of *modification operations* to the update workload. Recall that the original TPC-D specification only has insertions and an equal amount of deletions. However, in a real environment there are bound to be at least a few modifications, for example to allow corrections to existing entries.

Since we are interested in updates, we only used the two tables from TPC-D that have updates, namely `order` and `lineitem`. For *2VNL/1*, we chose the updateable attribute to be `orderstatus` for the `order` table, and `linestatus` for the `lineitem` table which are both of type `CHAR(1)`. Choosing just only one attribute as updateable is probably unrealistic, but we wanted to compare *MVNL* with the "theoretic" best case scenario for *2VNL*. To judge the performance of *2VNL* overall, one should take the average between the best case, *2VNL/1*, and the worst case, *2VNL/k*.

For all our experiments we used a SUN UltraSparc 1 model 170, with 256MB of main memory, running Solaris 2.5. We stored our database in a raw disk partition in order to by-pass any Unix buffering and averaged our measurements over multiple runs.

We ran a number of experiments of which we only present the most important ones because of space limitations. In the first two experiments we scale the database size and the update workload and measure the time each algorithm takes to complete the updates. The third experiment compares the slowdown that queries face when they are modified to support multiversioning under *MVNL* or *2VNL*. Finally, the last experiment, examines the speed of read-only queries when they run concurrently with the warehouse update algorithms.

4.1 Scaling the database size

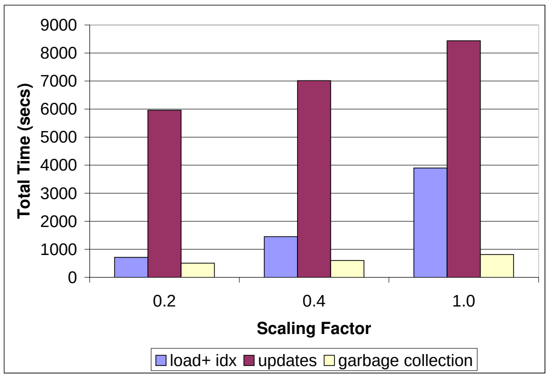
In our first experiment, we scaled the database size by changing the scaling factor of TPC-D. We tried three different scaling factors:

- **0.2**, which corresponds to 300K tuples in table `order` and 1200K tuples in `lineitem`

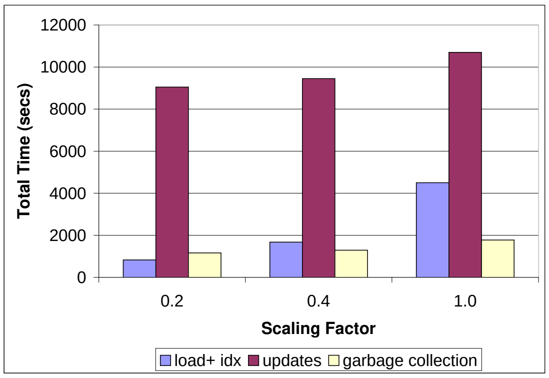
[7]One transaction contains all the operations on one particular order. It can contain for example the insertions on all tables that have to be made to successfully record a new order, or all the deletions that correspond to removing an order from the warehouse.

- **0.4**, which corresponds to 600K tuples in table `order` and 2400K tuples in `lineitem`
- **1.0**, which corresponds to 1500K tuples in table `order` and 6000K tuples in `lineitem`

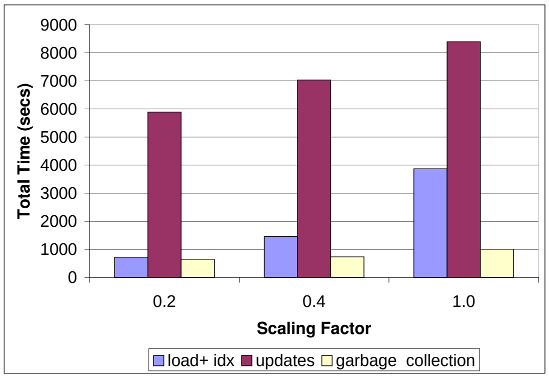
We kept the update workload constant at 150K maintenance transactions, split evenly among insertions and deletions. This workload corresponds to 50%, 25% and 10% of the database for scaling factors 0.2, 0.4 and 1.0 respectively. We also added a few modification operations (1% of the insertions) to make the update workload more realistic, since a real system should be able to handle corrections too.



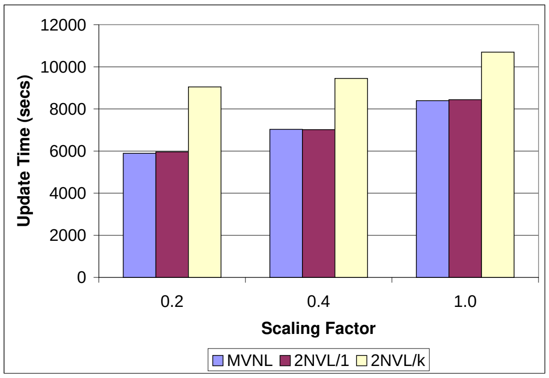
(a) *2VNL/1*



(b) *2VNL/k*



(c) *MVNL*



(d) All algorithms

Figure 6: Scaling the database size

In each experiment we first loaded the database from scratch and built indexes for the `orderkey` attribute on both `order` and `lineitem` tables (this is necessary to handle the deletions). Then we ran the update algorithm, and, performed garbage collection at the end. Although we report the total time to complete each of these phases, the time to complete the update phase is obviously the most important measurement.

Fig. 6a,b,c have the plots for each individual algorithm, for the different scaling factors. We can see that all algorithms scale really well. The time to perform the updates does grow with the size of the database, but unlike the loading & index creation phase, it is *not linear* in the size of the database.

Fig. 6d plots the time each algorithm takes to complete the updates, grouped by scaling factor. From this experiment, we find that *MVNL* is always faster or at least as fast as *2VNL/1* (the theoretic "best" case for *2VNL*) whereas *2VNL/k* (the worst case for *2VNL*) consistently takes much longer (54% for SF=0.2, 34% for SF=0.4 and 27% for SF=1.0). This can be explained by the fact that *2VNL/k*, because of the schema duplication, has to scan almost twice as much data as *MVNL* or *2VNL/1* which results in the degradation in performance.

4.2 Scaling the maintenance workload

In the next set of experiments, we kept the database size constant, but varied the update workload and compared the time each algorithm needed to perform the updates.

In the first experiment, we scaled the size of the update workload from 30K maintenance transactions to 150K, in 30K increments. In all cases, the workload was equally composed of insertions and deletions, and we also had a small percentage of modifications (=1% of the insertions). We report the time to complete the updates (insertions, deletions and modifications) for each algorithm. Fig. 7 has the results of our experiments, grouped by algorithm. For each algorithm the first column corresponds to the time to complete 30K updates, the second column to the time to complete 60K updates, etc. As expected, the time to complete the updates is always linear in the size of the update workload. That holds for all algorithms, although again we see that *MVNL* ties the "best case" for *2VNL*, whereas the worst case for *2VNL* is on average 36% slower.

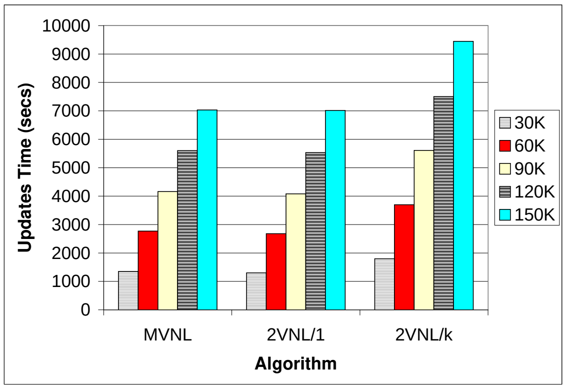
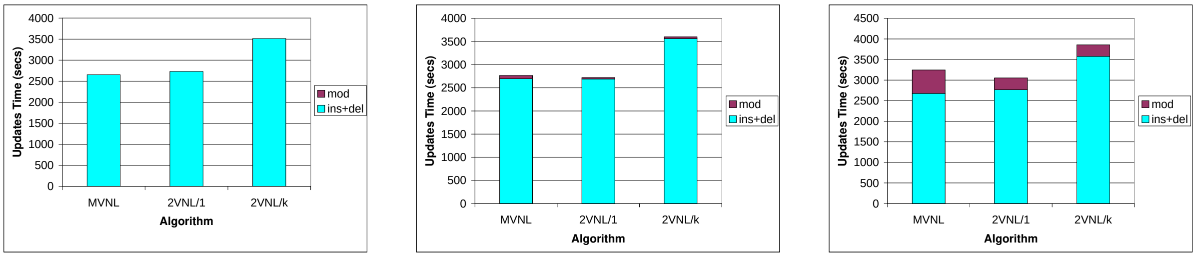


Figure 7: Scaling the maintenance workload

In the second experiment, we varied the percentage of modifications in the maintenance workload. At a scaling factor of 0.5 and an update workload of 60K maintenance transactions (about 8% of the database), we run three experiments, one with no modifications at all (Fig. 8a), one with 1% modifications (Fig. 8b) and one with 10% modifications[8] (Fig. 8c), reporting the total time to complete the updates for each algorithm.

[8]All percentages are based on the number of insertions, 30K.



(a) no modifications (b) 1% modifications (c) 10% modifications

Figure 8: Modifications as % of the insertions

These results illustrate the only "weak point" of update algorithms that use vertical redundancy: *modifications*. Since all modifications have to be translated into two physical operations (one to mark the previous value as deleted and and one to insert a new tuple with the new values), it is expected that workloads with a big number of modifications will be processed slower than with algorithms that employ horizontal redundancy (and perform modifications in-place). However, even having to perform two operations for every modification, we can see that *MVNL* is still faster than *2VNL/k*, and we believe this will also be the case with *n-VNL*. Overall, we expect that in typical systems the amount of modifications will be extremely small, for example TPC-D has none whatsoever, so this will not be a problem.

Running the update algorithms by themselves does provide some indication of their behavior, but to be able to better assess their performance we need to run them in conjunction with user queries. We present the results from these experiments in the next two sections.

4.3 Effect on queries: Schema changes

The online update algorithms require making changes to the relation schema and also force all user queries to be modified accordingly. These changes affect the response time of queries. In this series of experiments, we first ran the online update algorithms, and, after the warehouse has been refreshed, we executed some "benchmark" queries and measured their response times. We compare these response times to see which algorithm poses the most overhead to query execution. Since the algorithms were not running concurrently with user queries, this experiment should reflect slowdown on queries because of schema changes and query modifications mandated by each algorithm.

This series of experiments ran on a TPC-D database with scaling factor 0.5, whereas the update workload consisted of 75K maintenance operations (split evenly among insertions and deletions, with an extra 1% modifications), which correspond to roughly 10% of the warehouse. We ran two sets of experiments:

- one with *dynamic* queries, which accessed the portion of the warehouse that got updated, and,

- one with *static* queries, which accessed only parts of the warehouse that were not affected at all by the updates.

To get a representative query mix for each experiment, we also varied the query *selectivity*. We had three groups of queries, *low* selectivity queries (that return just 0.1% of the table = 750 tuples), *medium* selectivity queries (that return 1% = 7500 tuples), and *high* selectivity queries (that return 10% = 75K tuples). Furthermore, each query group was composed of two queries: one on the `order` table and one on the `lineitem table`. For every experiment, we report the total response time for each query group (averaged over multiple runs).

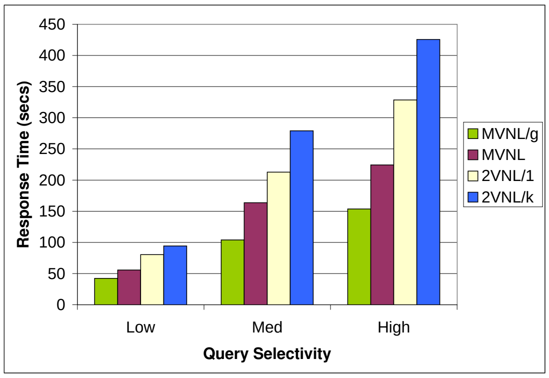


Figure 9: Dynamic queries

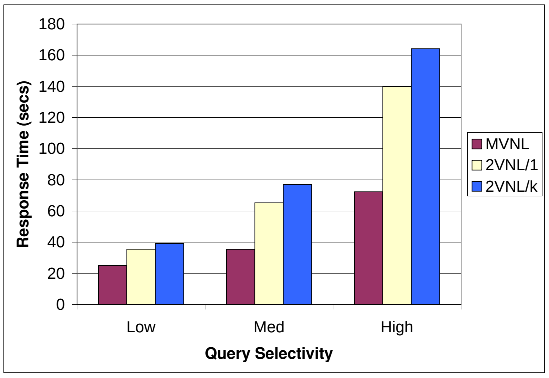


Figure 10: Static queries

Fig. 9 has the results of our experiments for dynamic queries, grouped by query selectivity. The first column corresponds to *MVNL* with the addition of a garbage collection phase (*MVNL/g*) before running the queries. The second column is *MVNL*, the third one *2VNL/1* and the last one *2VNL/k*. As we can see from the plots, in all experiments with dynamic queries, *MVNL* had the lowest response times, significantly lower (up to 30%) than the best case for *2VNL* and almost half the response time of the worst case for *2VNL*. *MVNL* after garbage collection was, as expected, an improvement over *MVNL* with 30% lower response times on average. The superiority of *MVNL* over *2VNL* can be explained by the fact that, while both algorithms add an extra qualification clause to filter out tuples that are supposed to be "invisible" to the query, *2VNL* must also choose at the tuple level which "version" of the attribute to return back to the query (using a CASE expression), which further delays each query.

Fig. 10 contains the results for static queries, grouped by query selectivity. Garbage collection will not influence the portion of the warehouse that is being accessed by static queries, so we did not include the case of *MVNL* after garbage collection in this plot. Moreover, all qualifying tuples in a static query should be "visible" to the queries, and hence, differences in response times are mainly because of differences in the relation schema and the evaluation of the CASE expression[9].

[9]Although the queries access tuples that have not been modified, query modification should have blindly included the CASE expression in all queries.

Indeed, our results show that *MVNL* is again the best of all alternatives, with significant "distance" from *2VNL/1* (30% faster for low selectivity queries, 46% faster for medium and 48% faster for high selectivity queries). The gap between *MVNL* and *2VNL/k* is even bigger, up to 55%, which is expected, since *2VNL/k* in effect doubles the relation size.

4.4 Effect on queries: Concurrent updates

We repeated the experiment of the previous section, but this time we let the queries run concurrently with the update algorithms. We used a mix of one low, one medium and one high selectivity query group (which, again, consisted of queries to both `order` and `lineitem` tables), and report the total response time for each query set (aggregated over multiple runs).

Our results for dynamic queries are in Fig. 11, where we run the queries while updating the warehouse using *MVNL*, *2VNL/1* and *2VNL/k*. We can see a clear "win" for *MVNL* (with queries running 24% faster than with *2VNL/1*), and a "dissapointing" slowdown for *2VNL/k* (with more than double the query response time of queries running concurrently with *MVNL*).

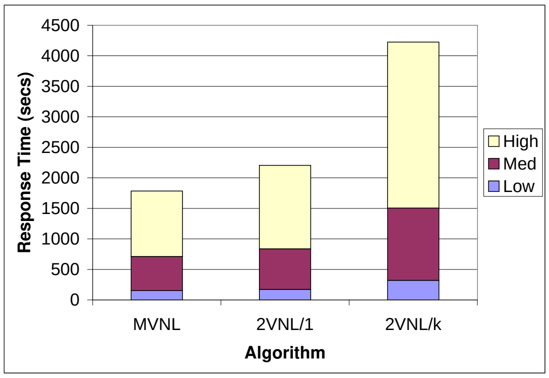


Figure 11: Concurrent, Dynamic queries

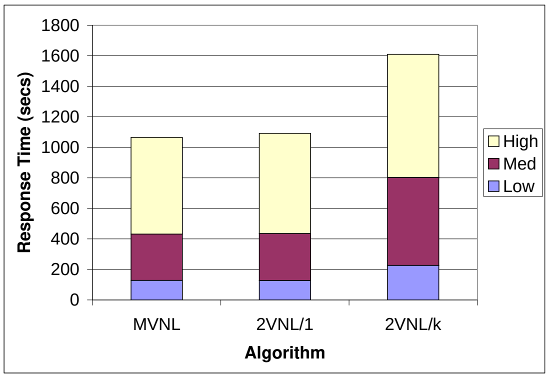


Figure 12: Concurrent, Static queries

The results from our experiments with static queries are in Fig. 12. This time, *MVNL* and *2VNL/1* produced similar query response times, whereas *2VNL/k*, as expected, is exhibiting really bad performance (50% slowdown compared to the other two).

By comparing the results from these two experiments we realize that while on the dynamic queries case, *MVNL* is exhibiting much better performance than *2VNL/1*, in the static queries case, *MVNL* and *2VNL/1* have similar performance. The reason behind this is that in the static case, there is no data contention between the updater process and the user queries, so any slowdown in the performance of queries comes mainly from the load on the warehouse server. In our first series of experiments (Sec. 4.1) we have established that both *MVNL* and *2VNL/1* take approximately the same time to complete the updates, which means similar server loads.

5 Conclusions

In this paper, we described an online warehouse update algorithm, *MVNL*, which uses multiversioning to allow the update process to run in the background while the queries execute concurrently and access a fully consistent version of the warehouse. *MVNL* employs *vertical redundancy* and stores new versions as separate tuples. We compared our algorithm to one that uses *horizontal redundancy* and stores the before-values of attributes by extending the relation schema. We have calculated the storage requirements for each algorithm and concluded that vertical redundancy is almost always more economical than horizontal redundancy.

We have implemented both algorithms on top of an Informix Dynamic Server and ran experiments using the TPC-D workload with scaling factor up to 1.0. We ran three series of experiments: running only the update algorithms and measuring the total time to complete the updates, running queries after the updates where performed and comparing the response time of the queries, and, running the update algorithms concurrently with queries and comparing the total response time of the queries. In all experiments, vertical redundancy exhibited significantly better performance than horizontal redundancy, with the exception of a few cases where the best case for horizontal redundancy matched the performance of vertical redundancy.

Overall, vertical redundancy is a more robust solution, since it needs no tuning and no knowledge of the application domain to implement it, unlike horizontal redundancy where identifying the updateable attributes is an important design decision. Its good performance and small storage overhead, make vertical redundancy the best choice for an online warehouse update algorithm.

Acknowledgements We would like to thank Yannis Kotidis, Manuel Rodriguez-Martinez, Kostas Stathatos and Bjorn T. Jonsson for many useful discussions, comments and suggestions. We would also like to thank Informix Software, Inc. for making available the Informix Universal Server through the University Grant Program.

Disclaimer The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government.

References

[AS93] Divyakant Agrawal and Soumitra Sengupta. "Modular Synchronization in Distributed, Multiversion Databases: Version Control and Concurrency Control". *IEEE Transactions on Knowledge and Data Engineering*, 5(1):126–137, February 1993.

[B+98] Phil Bernstein et al. "The Asilomar Report on Database Research". *SIGMOD Record*, 27(4), December 1998. (to appear).

[BBG+95] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. "A Critique of ANSI SQL Isolation Levels". In *Proc. of the ACM SIGMOD Conference*, pages 1–10, San Jose, California, June 1995.

[BC92a] Paul M. Bober and Michael J. Carey. "Multiversion Query Locking". In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 497–510, British Columbia, Canada, August 1992.

[BC92b] Paul M. Bober and Michael J. Carey. "On Mixing Queries and Transactions via Multiversion Locking". In *Proceedings of the Eighth International Conference on Data Engineering*, pages 535–545, Tempe, Arizona, USA, February 1992.

[BC97] Paul M. Bober and Michael J. Carey. "Indexing for Multiversion Locking: Alternatives and Performance Evaluation". *IEEE Transactions on Knowledge and Data Engineering*, 9(1):68–84, January 1997.

[BG83] Philip A. Bernstein and Nathan Goodman. "Multiversion Concurrency Control - Theory and Algorithms". *ACM Transactions on Database Systems*, 8(4):465–483, December 1983.

[BS83] Gael N. Buckley and Abraham Silberschatz. "Obtaining Progressive Protocols for a Simple Multiversion Database Model". In *Proceedings of the 9th International Conference on Very Large Data Bases*, pages 74–80, Florence, Italy, October 1983.

[CFL+82] Arvola Chan, Stephen Fox, Wen-Te K. Lin, Anil Nori, and Daniel R. Ries. "The Implementation of an Integrated Concurrency Control and Recovery Scheme". In *Proc. of the ACM SIGMOD Conference*, pages 184–191, Orlando, Florida, June 1982.

[CGL+96] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. "Algorithms for Deferred View Maintenance". In *Proc. of the ACM SIGMOD Conference*, pages 469–480, Montreal, Canada, June 1996.

[dBS96] Jochen Van den Bercken and Bernhard Seeger. "Query Processing Techniques for Multiversion Access Methods". In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 168–179, Bombay, India, September 1996.

[DDWJR98] Lyman Do, Pamela Drew, Vish Jumani Wei Jin, and David Van Rossum. "Issues in Developing Very Large Data Warehouses". In *Proceedings of the 24th International Conference on Very Large Data Bases*, New York City, USA, August 1998.

[JMR97] H. V. Jagadish, Inderpal Singh Mumick, and Michael Rabinovich. “Scalable Versioning in Distributed Databases with Commuting Updates”. In *Proceedings of the Thirteenth International Conference on Data Engineering*, pages 520–531, Birmingham, U.K., April 1997.

[LS89] David B. Lomet and Betty Salzberg. “Access Methods for Multiversion Data”. In *Proc. of the ACM SIGMOD Conference*, pages 315–324, Portland, Oregon, June 1989.

[LS90] David B. Lomet and Betty Salzberg. “The Performance of a Multiversion Access Method”. In *Proc. of the ACM SIGMOD Conference*, pages 353–363, Atlantic City, New Jersey, May 1990.

[LST97] Francois Llirbat, Eric Simon, and Dimitri Tombroff. “Using Versions in Update Transactions: Application to Integrity Checking”. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 96–105, Athens, Greece, August 1997.

[MOPW98] Peter Muth, Patrick E. O’Neil, Achim Pick, and Gerhard Weikum. “Design, Implementation, and Performance of the LHAM Log-Structured History Data Access Method”. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 452–463, New York City, USA, August 1998.

[MPL92] C. Mohan, Hamid Pirahesh, and Raymond A. Lorie. “Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions”. In *Proc. of the ACM SIGMOD Conference*, pages 124–133, San Diego, California, June 1992.

[MWYC96] Arif Merchant, Kun-Lung Wu, Philip S. Yu, and Ming-Syan Chen. “Performance Analysis of Dynamic Finite Versioning Schemes: Storage Cost vs. Obsolescence”. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):985–1001, December 1996.

[QW97] Dallan Quass and Jennifer Widom. “On-Line Warehouse View Maintenance”. In *Proc. of the ACM SIGMOD Conference*, pages 393–404, Tucson, Arizona, May 1997.

[Ree83] David P. Reed. “Implementing Atomic Actions on Decentralized Data”. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[Rou98] Nick Roussopoulos. “Materialized Views and Data Warehouses”. *SIGMOD Record*, 27(1), March 1998.

[SA93] O. T. Satyanarayanan and Divyakant Agrawal. “Efficient Execution of Read-Only Transactions in Replicated Multiversion Databases”. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):859–871, October 1993.

[SR81] Richard Edwin Stearns and Daniel J. Resenkrantz. "Distributed database concurrency controls using before values". In *Proc. of the ACM SIGMOD Conference*, pages 74–83, Ann Arbor, Michigan, April 1981.

[Sto75] Michael Stonebraker. "Implementation of Integrity Constraints and Views by Query Modification". In *Proc. of the ACM SIGMOD Conference*, pages 65–78, San Jose, California, May 1975.

[Sto87] Michael Stonebraker. "The Design of the POSTGRES Storage System.". In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 289–300, Brighton, England, September 1987.

[Tra98] Transaction Processing Council. *"TPC-D Benchmark Specification"*, February 1998.

[VV97] Peter J. Varman and Rakesh M. Verma. "An Efficient Multiversion Access Structure". *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, May 1997.