Integration of Time Versions into a Relational Database System

P. Dadam, V. Lum, H.-D. Werner

IBM Scientific Center Heidelberg
Tiergartenstr. 15, D-6900 Heidelberg, West Germany

ABSTRACT

New application areas for database systems, such as office automation and CAD/CAM will require to support not only access to the current data, as is done in current database systems, but also to previous instances of the data (versions). This means that time version support is needed. This paper presents the design considerations for a database system currently under implementation, that integrates time version support as a normal database function. It is shown that many subtle issues, such as choice of a suitable timestamp, how to store history data in a compact form, how to integrate version management into update processing, recovery, concurrency control, etc., have to be considered together to obtain an optimal design.

1. INTRODUCTION

The need to support the time domain within a database system has been indicated already several years ago (Bj75,Bu77,La73,La74). However, serious investigations by the database community have not been done until recently. The impetus for the recent actions originates from the growing interest in applying database technology to new applications. It so happens that in these applications the need to have time support is strong.

Although much work has been done recently (An82,CW83,Ki83,Kl81,KL83, MS83), it has been confined to conceptual studies in that the researchers concentrated on the effect of time from the users' perspective. To support time in a database system, one must have in the underlying support all the time information associated not only with data, but also with some actions executed on the data base (e.g. updates and corrections for walk-through-time queries). This, indeed, is the objective of this paper.

In the past there have been efforts to include into a system the function to support time versions. However, those studies have been mainly used for concurrency control purposes (BG83,BHR80,La82,PK84,Re78). Their approaches cannot be used to support the processing of history data that is needed for the new applications. To our knowledge, the only exception is Reed's system /Re78/ which, designed for concurrency control in a distributed environment, does permit to pose an "asof" (as of) query to some extent. His method, however, does not allow a user to pose queries which want to see the changes of some data items over time ("walk through time" queries) nor does the system attempt to obtain an optimal usage of its storage space.

The support of the time domain, as indicated later, cannot be effectively implemented on top of an existing database system. It must be done as an integral part of the design of the system. In a previous report (Lu84), we have discussed conceptually how a system can be designed to have the time domain support. In this paper, we will show that, while the conceptual design may seem to be straightforward, many subtle issues must be solved first.

In the next sections we will show first that there are many ways to store history data. We will analyze several possible alternatives and narrow down to the ones that seem to be most appropriate. We will discuss issues of choosing a timestamp and the effect of its selection strategy to performance. We will describe the effects of update operations to performance and show that one must analyze update strategies together with time version handling to determine a proper design. We will then conclude that the strategy of storing time information should be integrated into the design of concurrency control and recovery to arrive at an optimal solution.

2. THE EXTENDED RELATIONAL MODEL

Extending a relation (two-dimensional table) by time means to conceptually add a new dimension. The resulting data model can be viewed (Bj75) as a 3-dimensional cube (see fig. 1). The vertical slices represent the relation (table) at a specific point of time, with the front slice being the current one. Each horizontal slice represents a tuple and all its instances over time. Fig. 1, however, does not show the insertion and deletion of tuples after the initial load. A more realistic representation showing these effects is given in fig. 2. "Asof" queries logically work just on one vertical slice. "Walk through time" queries either work on a sequence of complete vertical slices, or on a sequence of selected parts of vertical slices ("rows"), depending on whether the logical view of having versions of relations or having versions of single tuples (see also section 5.2.2) is held.

3. ON TOP VS INTEGRATED TIME VERSION SUPPORT

At first sight, it seems to be very natural to implement time versions 'on top' of a relational database system in a view-like manner (Ch76). That means, the database system itself is not aware of supporting time versions. Time version support is therefore just an application from the system's point of view. This 'view' can be implemented by extending each relation to be versioned by two time attributes telling for each tuple when it has been created and when it has been logically deleted. A query pre-processor could help to implement the time 'view' (MS83,We83).

As an example, consider figure 3. To retrieve all EMP tuples which were valid at 01/25/82, the user can specify a query of the following form:

```
  SELECT *
  FROM EMP AS-OF 01/25/82.
```

A query preprocessor might transform this query into the form:

```
  SELECT *
  FROM EMP
  WHERE FROM-TIME ≤ 820125 AND TO-TIME ≥ 820125.
```

A great advantage of this approach is that time version support can be implemented rather quickly and easily on existing database systems with no change necessary. However, there are several drawbacks with this approach. For example, performance is decreasing with increasing number of versions, storage space is wasted, and the relational schema must not change over time (Lu84).

Performance degradation arises not only from large relations (tables) but also from indexing tables. Since all tuples are 'current' from the systems point of view, the indexes will contain not only information for the 'most current' tuples but also for 'old' tuples. As a consequence, indexes will grow not only by real insertions but also by updates. Other performance problems may arise from the resulting bad overall system architecture (e.g. doubling of functions, no global optimization, pre-processor data treated as user data, etc.) and that 'logical key uniqueness' has to be ensured now at the application layer (e.g., see ManNo in figure 3).

As logically old and logically current tuples are stored within the same table, both must also have the same structure. As a consequence, even if only one attribute value is changed from one version to another, both versions have to be stored completely. That is, the same unchanged attribute values are stored again and again, wasting much of the storage space.

For database systems which are designed to support (history) time versions, one must take into account that the number and types of attributes (the attribute structure) may change over time. Database systems of today, however, support, at best, only very restricted structural changes without data reorganization.

Hence the conclusion is clear: the performance, storage, and structural problems caused by an 'on top' implementation of versions are not tolerable. The best way to solve these problems is to integrate the time version support into the database system. How this can be done will be discussed in the following sections.

4. DESIGN GOALS FOR INTEGRATED TIME VERSION SUPPORT

As there are many choices to implement time version support, depending on one's objectives, one has to define first the primary goals. Our goals are to allow:

- Fast access to the current version.
- Many versions on-line.
- Selective versioning.
- Changes of the relational schema.

These goals are motivated as follows:

(1) Even in a system with time version support one can expect that most queries will work on the current data. This means, that access time to the current data can be expected to dominate the system's overall performance. Therefore, access to current data should be competitive to systems without time version support. Similar arguments hold for updates as well.

(2) If one wants to support on-line ad hoc queries which require access to history data, one must be able to keep a reasonable amount of versions on-line. Despite the fact that the development of new storage technologies (e.g. optical disks) offers new capabilities for storing large amounts of data, an extensive waste of

space for storing versions will still cause problems. Therefore one has to think about how to represent versions compactly.

(3) Supporting time versions will cost some price. However, the additional overhead created by having time versions must be relatively small; if some data is not versioned, then updates should be about as fast as in 'current view' database systems. This means that the system should also be suitable for environments where no (or only partial) time version support is required. We believe that it is very unlikely that everyone will want everything to be versioned. It seems to be more realistic to assume that some relations will be completely (all attributes) versioned, while others will have some attributes versioned, and others will not be versioned at all. Thus selective versioning has to be offered if such a system is to be accepted by users. (4) The same is true with structure changes as already discussed in the previous section.

5. DESIGN CONSIDERATIONS FOR TIME VERSION IMPLEMENTATION

5.1 VERSIONING STRATEGIES

In principle, one has the possibility to either store all versions of an object completely, or to store at least one version completely and the other ones as some kind of 'differences' or deltas (see also /We83/). The structure of deltas depends on the type of object one uses for versioning. We will discuss this in section 5.2.

Though time versions should be addressed via a timestamp and not a version number, we will use version numbers in the following discussion for convenience reasons. Denote $CV_x(n)$ version number n of object x, with n_0 being the oldest version. "CV" stands for complete version. Denote $\Delta_x(k,k')$ the delta (delta version) between the versions number k and k' of object x. If a version of an object is not stored as a complete version but only as a delta version, to materialize the version it has to be reconstructed using the delta version and the corresponding complete version. The following basic versioning strategies can be used for representing versions:

VS-1: versions(x) =
$(\Delta_x(n,n-1),\Delta_x(n-1,n-2),\dots,\Delta_x(n_0+1,n_0),CV_x(n_0))$

VS-2: versions(x) =
$(\Delta_x(n,n_0),\Delta_x(n-1,n_0),\dots,\Delta_x(n_0+1,n_0),CV_x(n_0))$

VS-3: versions(x) =
$(CV_x(n),\Delta_x(n,n-1),\Delta_x(n-1,n-2),\dots,\Delta_x(n_0+1,n_0))$

VS-4: versions(x) =
$(CV_x(n),\Delta_x(n,n-1),\Delta_x(n,n-2),\dots,\Delta_x(n,n_0))$

VS-5: versions(x) =
$(CV_x(n),CV_x(n-1),\dots,CV_x(n_0))$

VS-1 reads as follows: Version n_0 of object x is represented by a complete version. Version n_0+1 is represented by a delta version. To materialize version n_0+1, one has to apply this delta against version n_0 because the n_0+1 version is a (n_0+1,n_0)-delta. To materialize version n_0+2, one has first to materialize version n_0+1 and then to apply the (n_0+2,n_0+1)-delta, etc. We shall refer to a versioning strategy which has to reconstruct newer states from older states as forward oriented versioning strategy. The one which pursues the opposite direction is called backward oriented versioning strategy. As one can see, both VS-1 and VS-2 are forward oriented, while VS-3 and VS-4 are the backward oriented counterparts. VS-5 shows an equivalent versioning strategy based on complete versions as discussed in section 3.

These versioning strategies have the following characteristics: In VS-1, the current version is supported worst. That is, whenever the current version is to be accessed, it has to be materialized first using $\Delta(n_0+1,n_0)$, $\Delta(n_0+2,n_0+1)$, ... up to $\Delta(n,n-1)$ which requires n-1 iterations. On the other hand, the older a version is the faster will the access be. This is contrary to current practice, and therefore contradicts the goal just mentioned; hence we can exclude VS-1 from further investigations.

VS-2: Every version, and thus the current one too, can be materialized in at most two steps, because all deltas are related to the basic version. Access time to a version therefore is not dependent on its age. Hence VS-2 is certainly an interesting candidate.

VS-3: Access to the current version is supported without any additional penalty. All versions, except the current one, are expressed as deltas to the successor-in-time version. Hence access time to versions will grow with their age, and this seems to be acceptable, too. Therefore we will also keep an eye on VS-3.

VS-4: This strategy seems to be the ideal choice at first sight. Having as fast access to the current version as VS-3 and as fast access to older versions as VS-2. Unfortunately there is a significant difference to VS-2. While VS-2 deltas remain unchanged when new versions are created, VS-4 versions have to be recomputed completely whenever a new version is created. This results from the fact, that VS-4 deltas are related to the current version which changes whenever a new version is created. It's rather obvious that this overhead is not tolerable in general. Hence we can exclude VS-4 from further considerations.

Finally, we reject VS-5 because storage space requirement is prohibitive (see section 3). As a result, only VS-2 and VS-3 remain for further analysis.

5.2 UNITS OF VERSIONING

Versioning can be done for physical as well as for logical objects. Physical objects in our sense are files, tracks, and blocks (pages), while logical objects are the database, relations, and tuples. We shall discuss these two aspects next.

5.2.1 Versioning of Physical Objects

Suppose one chooses pages as the unit of versioning, called page versions. That is, a page is just considered as a byte or bit string without worrying about what the bits and bytes mean. A very useful method for this approach is the XOR bit representation /HR79/. Such an XOR version can be stored very compactly if the two pages differ only in a few bits, because in this case the resulting bit vector contains many zero bits and well-known compression methods for bit strings can be used for a compact representation (delta). However, as even little changes will cause the pattern in the page to change in a large section, it may be hard to get small deltas.

When using a version scheme based on page versions, versioning should be handled at the page handling layer of the database system. With the help of an appropriate timestamping scheme (see section 5.3), current and old page versions can then be logically addressed in a uniform way via <pageid,timestamp>. Using the VS-2 versioning strategy as introduced in the previous section, an update of the current version of page p giving page version p(n), would cause to fetch also $p(n_0)$, the basic (complete) version $CV_p(n_0)$ of page p. $\Delta_p(n,n_0)$ can be obtained by first computing p(n) XOR $p(n_0)$ and then applying some bit string compression technique to the result, if possible. Versioning based on VS-3 would be performed similarly. Instead of using the basic page version, however, p(n) XOR p(n-1) would be computed. The computational overhead for computing the deltas is higher for VS-2 because $p(n_0)$ has to be fetched additionally while p(n-1) will have been already fetched in most cases to compute p(n), the after-image. The storage overhead for VS-2 can also be expected to be somewhat higher because p(n) and $p(n_0)$ will normally show more differences than p(n) and p(n-1). Thus the probability to have an XOR bit vector with many zero bits giving the chance for a significant compression, will be higher for VS-3. Nevertheless both strategies remain applicable.

From a simplicity point of view, page versioning looks quite appealing. There are, however, some drawbacks because page versioning implies also page locking (or coarser). This is required in order to guarantee a correct order of timestamps within the versions of an object (similar to the 'lost update problem' /Gr78/; see also section 5.3.1). As a consequence, concurrency is potentially decreased compared to tuple locking. When used in combination with "long locks" (HL82) on logical objects which might share pages (which results in page locks on these pages), this scheme may even be not tolerable. The second drawback is that selective attribute versioning is not possible, because only uninterpreted bit patterns on pages are considered. As a consequence, the decision whether to version, will be at the tuple (relation) level. Such relations could be put into segments (files) or on pages with the same objective.

The drawbacks of having page versions (page locking, large deltas, very restricted selective versioning) seem to be too serious. In the following we will focus therefore on versioning of logical objects only.

5.2.2 Versioning of Logical Objects

We start to discuss versioning of logical objects on tuple level, called tuple versions in the following. Analogous to page versions, tuple versions should be handled at the tuple handling layer of the database system. The versioning itself can be done similarly to page versioning with the help of XOR differences to describe the difference between the before- and the after-image of a given tuple. Instead of that "physically" oriented scheme, one can also use the operations performed on the different attributes of the tuple to describe the change (Bj75), thus implicitly describing the difference. This approach is called logical versioning in the following.

If attribute values are rather long, e.g., if an attribute is a "long field" (HL82), it may be appropriate to apply the XOR technique also at the attribute level. Another approach to deal with such attribute types is to provide partial field operations in addition to the normal field operations which treat the field (the attribute value) always as a whole. Such operations can be "update a field partially", or, if the field is of variable length, "insert a part of the field "or "delete a part of the field". Obviously, these partial field operations can also directly be used for logical versioning.

Hence, versioning of logical objects allows to choose among a variety of methods in order to apply the optimal delta method such that the resulting delta is as small as possible. Also selective versioning is possible now, because the logical unit 'tuple' consisting of 'attribute values' is known now to the layer at which versioning is performed. Let us now analyze logical versioning with strategies VS-2 and VS-3:

Versioning strategy VS-2:

Let x_n be the current before-image version, stored as $\Delta_x(n,n_0)$ of object (tuple) x, and x_{n+1} the new after-image version to be created and stored as $\Delta_x(n+1,n_0)$. Let version number n_0 be again the basic version stored as $CV_x(n_0)$. Using strategy VS-2, there are two ways how one can compute a new delta version. One way, in the following called VS-2/1, is to compare directly the temporarily materialized after-images of versions number n+1 and n_0 (essentially a physical comparison). In this case, however, only very restricted selective versioning as with page versions is possible. Another way, in the following called VS-2/2, is to compute first $\Delta_x(n+1,n)$ and then to combine or "merge" $\Delta_x(n+1,n)$ and $\Delta_x(n,n_0)$ such that $\Delta_x(n+1,n_0)$ is obtained.

Applying VS-2/1, the tuple and attribute oriented operations cannot be used for delta purposes. That is, logical versioning is not possible. In principle only XOR deltas based on the before and after-image of the whole tuple or of the attributes to be versioned can be computed. Applying VS-2/2 instead makes delta creation more complex, but offers the potential chance to compute a more compact delta based on logical versioning, where appropriate.

However, even if VS-2/2 is used, there are still some problems in computing compact deltas. Consider, for example, the following case: A variable long attribute value has length zero (is 'empty') at insertion time and then 'grows' later on. VS-2/1 as well as VS-2/2 will always have to store the attribute's complete after-image whenever this attribute is updated regardless whether only one byte or the whole attribute value has changed from one version to the other. In such and similar cases VS-2 may waste a lot of storage space.

Another problem is that the basic version has to be kept 'for ever', because all delta versions are related to the basic version. This can be circumvented rather easily by creating from time to time a complete version instead of a delta version and to use that as reference basis for the following versions. This may, in some cases, also help to solve to some extent the problem with 'growing' fields just described above.

However, this would not solve the problem of purging history data. Generally, the purging of history data requires the re-calculation of most, if not all, of the deltas. This happens because the time at which complete versions of tuples are realized will differ from tuple to tuple and from the time beyond which data is to be purged. (Purging history data is a non-trivial problem and is beyond the scope of this paper.)

Versioning strategy VS-3:

This strategy provides fast access to the current version without any additional overhead. As the deltas are computed on the basis of the before-image of a tuple and the operations applied against it, or by comparing before and after-image of a tuple, respectively, one has all the flexibility of choosing the appropriate delta method as indicated before. As one saves the additional access to the oldest tuple, update can be performed faster in general. In addition, there is no need to keep the oldest tuple 'forever' which makes purging or off-loading of old versions somewhat less complicated. Of course, access time to old versions will grow proportionally to the number of versions one wants to go back in time. From the user's point of view this is probably what he expects.

Thus, we come to the conclusion that VS-3 is the most appropriate overall strategy and in the following we will therefore concentrate only on this strategy.

Up to now we have discussed how tuple versions can be implemented. A user, interacting with such a system with time version support might not think in versions of tuples but in versions of relations or may even have the view of having versions of the whole database. In the remainder of this section we will discuss how the view of having versions of relations can be supported. We restrict our discussion here to the case of a relation scan where the tuples have to be found via a physical scan through all the segment pages (segment scan). To simplify the discussion we will assume that only the current version of each object is stored in the (current) database and that all other versions are stored separately in a so-called history pool.

Assume a query of the form "Select * from EMP asof t_1" is posed. This would result in scanning all the segment pages where EMP is stored to get all tuples belonging to EMP, and comparing their timestamps with t_1. If a tuple's current timestamp is too 'new', one has to check whether there exist deltas of this tuple in the history pool whose timestamps satisfy the time condition. Hence, insertions do not cause any major problem. But what about deletions? Obviously we may not simply erase a tuple along with all its deltas if we want to have time version support. One solution would be to erase it in the database and to keep the deltas (including the version from the database) in the history pool. In this case every segment scan caused by a history query would enforce to scan the database segment as well as the history pool. A better solution is to keep a small remainder of the original tuple in the database which serves as a deletion mark and indicates that there existed a tuple and when it has been deleted. With a relatively small amount of additional storage space we can thus avoid unnecessary accesses to the history pool. In this way, versions of relations can be implemented via the concept of tuple versions, too.

5.3 TIMESTAMP SELECTION

5.3.1 Timestamp Properties

When we speak about 'time' in the following, we always mean the physical time which is related to and derived from the system's internal clock. (A detailed discussion about 'logical' (= user defined) and 'physical' (= system defined) time and their relationship can be found in /Lu84/.) As we deal with time versions, objects and versions of an object should be addressed via <object_id,time>. The question, however, is which time to associate with an object or version, since a transaction spans a period of time. For example, a transaction T has started at 10:17:10 and ended at 10:23:12. Which timestamp(s) may one choose for objects/versions created by T?

In the following, we want to analyze the restrictions which a <u>timestamp selection scheme</u> (TSS) has to obey to. Denote BOT_i and RtC_i begin of transaction T_i and the Ready-to-Commit point (the logical end) of transaction T_i, respectively. Denote further TS(x) the value of the timestamp assigned to an object x or, if x is an event like BOT or RtC, the point in time when that event occurred. As the timestamp assigned to a new version x_k of an object x created by transaction T_k should be related to the time interval in which transaction T_k has been executed, $TS(x_k) \in [TS(BOT_k),TS(RtC_k)]$ should hold. A TSS obeying this property is called <u>transaction-consistent.</u>

The question arises, whether one may use different timestamps for different objects within one transaction. Consider a banking account example. Given an account A with single positions a_i, i = 1,2,..,n, and an account B with single positions b_j, j = 1,2,...m. Assume, the integrity constraint

$$\sum_{i=1}^{n} value(a_i) - \sum_{j=1}^{m} value(b_j) = 0$$

is implicitly modelled by a transaction T_u. No value of an $a \in A$ may be changed without also changing the value of a position $b \in B$ within the same transaction in the same way. Assume, the tuple containing an $a \in A$ is updated first, getting timestamp t_1 assigned to it, and then the tuple containing a $b \in B$, getting timestamp t_2 assigned to it, such that $t_1 < t_2$. That is, TS(a) < TS(b) holds, too. If later on a query is executed which requests to see the (logical) state of the database at time t, $t_1 < t < t_2$, it may see an incomplete booking. To avoid this inconsistency, all objects created or updated by a transaction T_u (call this set W_u (<u>write set</u>) for short) must get the same timestamp. A TSS which obeys this property is called <u>atomicity-preserving.</u>

Still one has the question, which timestamp to choose within the given restrictions (transaction-consistent, atomicity-preserving). Obviously, to use always the RtC timestamp would be a legal choice. As we will see in section 5.3.2, this choice has also some drawbacks one might not want to have. We will therefore analyze the freedom one has in the choice of a "legal" timestamp.

It seems reasonable to require that the sequence of versions of one object, ordered by time, reflects also the logical order of updates applied against that object. If one would, for example, always choose the BOT timestamp of a transaction to timestamp all its created objects and versions, this requirement would not always be satisfied. Consider two transactions T_1 and T_2 which are executed in parallel. T_1 is started at time 100, reads and updates a tuple τ, say, new value 1000, at time 200, and terminates at time 250. T_2 is started at time 110, reads and updates tuple τ, say, new value 500, at time 120 and terminates at time 150. Hence, the sequence of after-image values, reflecting also the equivalent serial execution order, is 500 (created by T_2) followed by 1000 (created by T_1). The corresponding sequence of timestamps, however, would be 110 (T_2) followed by 100 (T_1) if the BOT timestamp would have been used. Thus, either the timestamps or the values of the deltas are not in proper order.

Using some insights and notions from concurrency control theory on serializabilty (BSW79,Da82, Pa79), we can formulate the problem more precisely: Consider two transactions T_1 and T_2 which have been executed in parallel, in symbols $T_1 \parallel T_2$, producing result $\Re$. If they have been correctly synchronized, then either the serial execution order T_1 before T_2 or vice versa, in symbols $T_1 < T_2$, or $T_2 < T_1$ respectively, would have produced the same result $\Re$, too. If only $T_i < T_j$, $i,j \in \{1,2\}$, $i \neq j$, would produce $\Re$ but not $T_j < T_i$, then one calls T_j to be <u>dependent</u> on T_i, in symbols $T_i \rightarrow T_j$. To keep the order of timestamps in accordance to an equivalent serial execution order, the following condition must always be met: $T_i \rightarrow T_j$, $i \neq j \Rightarrow TS(x_i) < TS(x_j)$. A TSS which obeys this property is called order-preserving.

In the following we assume that strict two-phase locking (2PL) is applied. That is, all locks are held until the Ready-to-Commit point (RtC), which is the logical end of the transaction. Denote U_k, $U_k \subseteq W_k$, the update set of T_k (= W_k without newly created objects) and $OTS_{max,k}$ the maximal (old) timestamp value of all objects accessed by transaction T_k for update, where $OTS_{max,k} := TS(BOT_k)$ if $U_k = \emptyset$. Denote further x_k again the newest version of an object x created by transaction T_k, and W_k transaction T_k's write set.

Definition:
A TSS is said to be legal if and only if it is transaction-consistent, atomicity-, and order-preserving.

Lemma 1:
A TSS is legal $\Leftrightarrow \forall T_k: W_k \neq \emptyset, \forall x \in W_k$:

1. $TS(x) \in TSI_k :=$
   $[\max\{OTS_{max,k}+\delta, TS(BOT_k)\}, TS(RtC_k)]$, $\delta > 0$,

and

2. $\neg\exists (x,y), x,y \in W_k: TS(y) \neq TS(x)$.

We call TSI_k to be the timestamp interval of transaction T_k.

Proof: (Sketch)
"$\Rightarrow$": To be transaction-consistent $TS(x) \in [TS(BOT_k), TS(RtC_k)]$, $x \in W_k$, must hold. To be order-preserving, $\forall x, \forall y$: $TS(x) > OTS(y)$ ($x \in W_k$, $y \in U_k$), must hold, too. Hence the lower bound of T_k's timestamp interval has to be set to $\max\{OTS_{max,k}+\delta, TS(BOT_k)\}$. As a "legal" TSS must be atomicity-preserving, condition 2 of lemma 1 is implied as well.
"$\Leftarrow$": Obviously, if $TS(x) \in TSI_k$, then TSS is transaction-consistent and order-preserving. To be atomicity-preserving as well, only one timestamp may be selected per transaction. This is guaranteed by condition 2 in lemma 1. Hence "$\Leftrightarrow$" holds. □

Lemma 1 can directly be used for designing version creation strategies as will be shown in the following section.

5.3.2 Assigning Timestamps to Objects

Assigning timestamps to objects has several aspects (see fig. 4). From the discussions above we know that only the RtC timestamp is really 'safe' because its property to be "legal" is not dependent on the timestamps of objects the transaction accesses for update. Hence every timestamping which is done before RtC can only be done tentatively and might have to be corrected at RtC. One could choose the BOT timestamp, or the timestamp according to the first write operation, or one could try to estimate the RtC timestamp. A method using tentative timestamping is outlined in section 5.3.3.

Final timestamping requires that the final (legal) timestamp is already known at timestamping time. This can be achieved either by timestamping the versions after RtC (see 'deferred update' in section 5.4.2), or via indirect timestamping. Direct timestamping means to put the timestamp physically into the object (e.g. as part of the tuple header). Indirect timestamping means that not the timestamp itself is stored in the object but only some kind of reference number (rn), e.g. a unique transaction sequence number. The relationship between this reference number and the corresponding timestamp is implemented via an rn-TS-table. As the timestamp can be assigned at RtC, there is no need to refetch any object for timestamping reasons. On the other hand, as objects in the current database can become very old without being updated, the rn-TS-table may become very large. Especially in cases where concurrency control also uses timestamped versions (e.g., in /Re78/), accesses to this table may become a bottleneck.

5.3.3 Optimistic Tentative Timestamping Scheme (OTTS)

Lemma 1 can be used to design a tentative timestamping scheme which may be interesting if access conflicts can be assumed to be rare ("optimistic" assumption). This scheme can be briefly sketched as follows: Before the first object is updated by transaction T_k, it chooses a timestamp $TS_k \in [TS(BOT_k), \text{"current-time"}]$. As long as no object y is accessed by T_k for update with $OTS(y) > TS_k$, all objects are timestamped with TS_k. If such a conflict occurs, however, TS_k is set to a new value (e.g. the current time), such that $TS_k > OTS(y)$. To be able to restamp tuples when necessary, all objects timestamped by T_k are recorded a list together with their (tentative) object timestamp $TS(x_k)$. At RtC time the object list is scanned for objects x_k with $TS(x_k) \neq TS_k$. These (and only these) objects are refetched and timestamped with TS_k. If no exceptions occurred, no objects have to be refetched and no additional overhead occurs as compared to systems without time versions. In the worst case when a conflict occurs while accessing the last object, all objects, except the last one, have to be refetched for re-timestamping. Hence this scheme is only appropriate when conflicts are expected to occur seldom.

Applying OTTS means to assign, from the user's point of view, rather arbitrary timestamps to transactions and to objects. One transaction may get its BOT timestamp, another one something in between BOT and RtC, and still another one its RtC timestamp to timestamp its objects. If transactions are rather short, this should cause no acceptance problems in general. If transactions may run rather long (e.g., some hours), OTTS generated timestamps may cause acceptance problems because the timestamps generated by OTTS may look too "arbitrary" to some users. So, even if the optimistic assumption is fulfilled, pure OTTS might therefore not be applicable in all cases.

5.4 INTEGRATING TIME VERSIONS INTO UPDATE PROCESSING

Versioning has to be integrated carefully into update processing in order to keep the additional overhead as small as possible. As we have restricted our analysis to versioning of logical objects (see section 5.2.1), we need not consider update processing schemes which are based on physical objects (e.g. shadow page technique /Lo77/ or database cache /Ba83,El82/). In the following we will discuss how VS-3 can be integrated when update processing is based on immediate update and when it is based on deferred update.

5.4.1 Immediate Update

Immediate update means that whenever an object is created or modified it is immediately propagated to the database (not necessarily forced immediately to disk). Immediate update is an 'optimistic' strategy in the sense that one expects transaction backout to occur rather seldom compared to the number of successfully completed transactions. It means also that the objects are not refetched in the commit phase (in contrast to the deferred update approach described below). This behavior, however, causes a problem with assigning the timestamps. Assume transaction T_k updates an object x giving version x_k. From lemma 1 we know that the timestamp must be chosen such that $TS(x_k) \in TSI_k$, and that this timestamp must be equal for all versions which T_k creates. As the lower bound of TSI_k, however, is dependent on the timestamps of objects which T_k accesses for update, this lower bound might not be known when creating x_k. Hence either tentative timestamping (e.g. OTTS) or indirect timestamping has to be used to solve this problem.

Let us now discuss a method of version creation, based on strategy VS-3, integrated into update processing. Let x(n) again denote the current version of object x. Using versioning strategy VS-3 the first update of object x would require the following steps:

1. Access $CV_x(n)$.
2. Compute $CV_x(n+1)$ and $\Delta_x(n+1,n)$.
3. Insert $\Delta_x(n+1,n)$ into history pool.
4. Replace $CV_x(n)$ by $CV_x(n+1)$ in the database.

As a result of repeated updates of x within the same transaction, VS-3 would produce several before-image deltas (step 3) because it is backward-oriented and therefore dependent on the most current version. Having several deltas for the same object (tuple) is not very desirable because space is wasted and processing of history queries would be more inefficient. To get at most one new delta per object (tuple) for a given transaction, step 2 has to be modified as follows:

Denote $x(n+1^i)$ the i-th after-image of version n+1 of x after i ($i \geq 0$) repeated updates within the current transaction, with $i = 0$ being the first update, and $\Delta_x(n+1^0,n)$ the first delta for x created by this transaction. Assume, the 2nd update is performed against object x. The new delta must comprise the effects of the first update ($\Delta_x(n+1^0,n)$ as well as the "difference" between the first and the second update $\Delta_x(n+1^1,n+1^0)$. Hence we can re-formulate step 2 as follows:

2'. Compute $CV_x(n+1^i)$.

If $i = 0 \rightarrow$ compute $\Delta_x(n+1^0,n)$.

If $i > 0 \rightarrow$ compute $\Delta_x(n+1^i,n) :=$

$$\Delta_x(n+1^i,n+1^{i-1}) \text{ "+" } \Delta_x(n+1^{i-1},n).$$

As the final delta version is being built "incrementally" (each update will add something to it) we call this approach <u>incremental versioning</u> and $\Delta_x(n+1^i,n+1^{i-1})$ an <u>incremental delta.</u>

5.4.2 Deferred Update

Deferred update means that modified and newly created objects are kept 'outside' the database and local to the corresponding transaction (in some kind of private workspace) until RtC. When the updates/inserts are committed, they are brought into the database in a separate write phase. This approach has been first proposed for distributed databases (Ro80,Th76,Th79) where updates may be prepared at another site than the objects are stored, to guarantee fast transaction backout, and where it is important to have a repeatable commit phase. Recently it has also been proposed for centralized systems, e.g. for 'optimistic concurrency control' OCC (KR81); also the "database cache" approach (Ba83,El82), although it is page-oriented, can be seen under this perspective, to some extent. Applying deferred update, only little changes are required for the version creation scheme just described. Instead of writing the deltas or after-images in steps 3 and 4 to the database or the history pool, respectively, they are written into the workspace. A separate write phase (step 5) has to be added to propagate the deltas and the after-images to the database and to the history pool.

The 'write phase' can be used to timestamp the objects 'on the fly' with the RtC timestamp without creating any additional overhead for this purpose. On the other hand, the deferred update approach can cause a significant additional overhead compared to immediate update if it is not integrated very carefully into the overall concurrency control and recovery concept. To use deferred update without further integration only for timestamping reasons may therefore not be acceptable in general (for further discussions see section 6.2).

5.4.3 Integrating Time Versions into Update Processing - Conclusions

In this section we have analyzed how time version (delta version) creation can be integrated into the normal update processing. Although the algorithms to compute compact deltas might look somewhat involved, the integration causes no real problem. In the next section we will re-consider both update processing strategies under the aspect of using time versions for concurrency control and recovery.

6. USING TIME VERSIONS FOR CONCURRENCY CONTROL AND RECOVERY

Recently several concurrency control (CC) methods have been proposed which make use of before-images to enhance concurrency between conflicting transactions (BG83,BHR80,Ch82,La82, PK84,Re78). As "before images" are a special kind of time version, it seems reasonable to use the time versions as described above for this purpose too. (We will abbreviate these version-based CC methods with VCC in the following.) Whether this is really reasonable or not shall be discussed in the following. Again, we restrict our considerations to strategy VS-3.

If a read-only transaction, called "reader" in the following, and an update transaction, called "writer", are executed in parallel, CC will often force the reader to wait for the writer to finish because the writer is updating an object which the reader wants to read. In some cases, however, it would cause no consistency (serializabilty) problem to give the previous version (before-image) to the reader instead. This before-image version often exists anyway, as it is needed for recovery purposes. The new idea of these VCC methods is to make these before-images available for CC too, and to define CC rules to decide whether an old version may be used (and sometimes also which version) or whether access to the current version is mandatory.

In all proposals, the versions used for VCC are the same as used for recovery purposes. That is, the CC versions are complete before-image versions. In the following we will analyze whether we could utilize our delta versions for this purpose as well. To simplify the discussion, we shall assume that new versions are written to the same physical and/or logical location where the previous version was stored. That is, objects (tuples) are updated in place. The substitution of the previously current version by the new current version takes place either immediately or deferred, depending on the update processing strategy. As a consequence, once the update is propagated to the database, the before-image version is no longer available because it has been overwritten by the new version.

Using delta versions for VCC purposes means that such a version, before it can be used, has to be materialized first. As complete versions are required, selective versioning obviously causes a problem because only the versioned attributes would show the correct old value in general. To solve this problem, one could compute a temporary complementary delta together with each regular delta which contains all attribute changes not covered by the regular delta. As a consequence, however, version creation, propagation of deltas and after-images, and materialization of before image versions become slightly more difficult.

Having complementary deltas where necessary is equivalent - from the CC point of view - to having no selective versioning. That is, all attributes are versioned, at least temporarily. In the following analysis we will assume complete versioning.

6.1 IMMEDIATE UPDATE

Immediate update, as described in section 5.4.1, means to create the new after-image directly in the database and to have a delta (before-image delta) in the history pool, showing which changes have to be applied against this after-image to get back the initial before-image. As this delta depends on the value of the current version, each update of the current version (the after-image) will cause the delta to be re-computed if one transaction shall create at most one delta per object or tuple (we called this "incremental versioning" in section 5.4.1) In other words, the before-image delta is not stable in general. As the after-image is not stable in general either, one has always to ensure that the after-image and the delta one wants to use for materializing the before-image version do fit together. As a consequence, propagating the after-image and the corresponding delta to database and history pool must be done in one step requiring some kind of short-time locking, in general, to ensure consistent input for readers. In this way, delta versions could be used for VCC too.

Let us now consider the usability of our deltas for undo-recovery purposes, i.e., whether they can substitute an undo-log. As an undo-log is used to remove the effects of incomplete transactions from the database (transaction abort, crash victim), log entries must be written to permanent storage before the corresponding change in the permanent database is performed (write-ahead-log principle /Gr78/). This means in our case, that the deltas must be written to the history pool before the after-image is written to the database. In addition, the deltas must always show what has to be done with the corresponding database tuple on disk, to obtain the before image value it had at BOT. We shall first assume, that incremental versioning is applied (see above).

Consider fig. 5. Assume that one transaction updates object x twice. Before the new after-image is written to disk, the corresponding delta is written to the history pool. If a crash occurs at the time indicated in this figure, the history pool delta could not be used to restore the database object to its old value it had before BOT. One can show, that this problem could be resolved by using a list of incremental deltas with update counters, instead of computing and storing only one delta.

Hence we can conclude that, if immediate update with incremental versioning (at least one delta per object and per transaction) is applied, then these deltas can be used for VCC but not for recovery purposes. If the incremental deltas are stored explicitly, e.g., as a linked list and some additional information like an update counter, they could also be used for recovery purposes. Since this means to waste additional storage space and to enhance the overhead for processing history queries, this may only be acceptable if repeated update occurs rather rarely.

6.2 DEFERRED UPDATE

In this approach, all after-images and corresponding deltas created by a transaction T_k during run-time are withheld from other transactions until RtC_k by simply keeping them in T_k's workspace. Hence, before RtC_k no before-image materialization is needed. When the write phase has started, however, synchronization of after-image and delta write is required and access to the before-image requires materialization as in the immediate update approach. As incremental versioning is done in the workspace, not visible to CC and other transactions, synchronization is needed only one time per object/delta and transaction. Thus, less overall overhead is necessary to guarantee consistent input for readers.

As only one after-image and one delta is written to the database after RtC, the "incremental versioning problem" as outlined above, does not exist here. One delta is sufficient to capture all undo-information necessary for crash-recovery (transaction-backout is directly handled with the help of the workspace; see below). Hence, deltas based on deferred update are able to substitute an undo-log.

A probably better idea, however, would be to make undo-recovery in the database superfluous by using the workspace in a permanent storage device to make the transaction's write phase repeatable. That is, to use the workspace as a transaction oriented redo-log. In this case, to undo a transaction which has not yet reached its RtC would mean to simply "forget" the workspace. On the other hand, if a crash occurred during the transaction's write phase, the workspace could be used to repeat it once more.

6.3 USING TIME VERSIONS FOR CC AND RECOVERY - CONCLUSION

Time versions (delta versions) can be used to implement version-based CC for both update processing strategies, immediate and deferred update. Their usability for recovery, however, is restricted to deferred update processing in general. The deferred update approach offers a very good basis (considering also the RtC timestamping aspect (see section 5.4.2)) for integrating time versions into update processing, concurrency control, and recovery.

7. SUMMARY AND CONCLUSIONS

Integrating time versions into a database system poses a lot of questions. For example, which object to choose (physical or logical), how to store the versions (complete, compact, forward, or backward oriented), how to select and assign an appropriate timestamp, how to integrate versioning into update processing, etc. These questions have been analyzed in this paper and solutions have been proposed. We have defined and discussed five basic versioning strategies. One of them has finally been selected for further investigation. It allows access to the current version without any additional overhead compared to systems with no version support. In addition it offers a good potential for creating small versions. The price to have compact versions is to have an increasing processing overhead for history queries according to the time one wants to go back.

The goal of this paper was to show that integrated time version support does not necessarily cause an 'exotic' system architecture. We therefore have analyzed how to integrate version creation into two different update processing strategies, immediate and deferred update, and whether, and under which conditions, these time versions (deltas) could also be used for concurrency control and recovery purposes. We have shown that a fully integrated approach causes no major problems for both update processing strategies, although it might not be a really good idea to use time versions for recovery purposes when immediate update is applied. Opposed to that, a full integration of time versions into the deferred update approach looks very promising.

Because of space, we had to restrict our analysis in this paper to some major points. For designing a system with time version support, other important issues have to be solved as well.

One important point is how to support structural changes. By storing the catalog as a relation, the versioning mechanism can be used to have versions of the catalog as well. However, additional problems like providing and defining the semantics of additional null values and host-language coupling have to be solved in this context, too.

Another important point is index support for history data. The general problem and possible solutions have been already discussed in another paper (Lu84). We only want to outline here, how versions of pointer lists (as they may occur in a "history index") can be stored in a rather compact way. As already mentioned in section 5.2.2, one uses partial field operations to obtain compact deltas for long attribute values. Treating a pointer list as a tuple of a unary relation with a variably long attribute value, an update or insertion at the object level causing a pointer value to be put into a pointer list, could be simply expressed as "insert value x after position y in field z". Obviously, this allows potentially to derive reasonably small deltas also for pointer lists.

As delta creation is automatically triggered by updates, and as updates may be incorrect (e.g., typing errors), the generated deltas, seen from the the user's point of view, may be incorrect too. Hence, one has also to deal with correction of errors. Obviously, to simply overwrite wrong values is not acceptable in general. As corrections are "events" one might asked for in queries, a system with time version support should be able to support three different types of 'history' queries. Queries which work on the corrected data, queries which work on the non-corrected data, and queries which answer when and which corrections have been issued. Another problem in this context is, how to implement a meaningful user interface to perform corrections. This problem has not been resolved yet.

Last, but not least, one has to deal with logical time. That is, timestamps for objects which a user may define and possibly also may want to change. Logical time is of special interest when the time at which the update is performed differs from the time when the change shall become logically "effective" (e.g., a retroactive raise in salary or new prices which shall become valid some days later). Providing logical time, which implies "change the history", it will probably be necessary to reproduce the sequence of updates (changes) on demand, providing a facility which allows to see which value did really (without subsequent "corrections") exist at a given point in time in the history. Hence, logical timestamps will, in general, not replace but complement the physical timestamps described in this paper (for further discussions see /Lu84/).

For simplicity we have restricted our discussion to relational database systems. As a matter of fact, the results presented here, have been derived from our design considerations for a database system which is supposed to provide a basis for the support of both, the hierarchical and the relational data model. This system, currently under implementation within the database project at the Heidelberg Scientific Center, is expected to be useful to many scientific, engineering and office applications. Some other aspects of this database system have already been reported in /DPS83,Ja84,JS82,SP82/.

Acknowledgement

We want to express our special thanks to R. Erbe and N. Staab who participated in the design and implementation of the tuple handling and time version management component of our database prototype. Many thanks also to J. Guenauer, P. Pistor, G. Walch, and J. Woodfill for helpful discussions.

8. REFERENCES

An82 Anderson, T.L.: Modeling Time at the Conceptual Level. Proc. Second Int. Conf. on Databases, Jerusalem, June, 1982

Ba83 Bayer, R.: Database System Design for High Performance. Proc. IFIP 83, Paris, Sept. 1983, pp. 147-155

BG83 Bernstein, Ph.A., Goodman, N.: Multiversion Concurrency Control - Theory and Algorithms. ACM TODS, Vol. 8, No. 4, pp. 465-483

BHR80 Bayer, R., Heller, H., Reiser, A.: Parallelism and Recovery in Database Systems. ACM TODS, Vol. 5, No. 2, 1980, pp. 139-156

BSW79 Bernstein, Ph.A., Shipman, D.W., Wong, W.S.: Formal Aspects of Serializability in Database Concurrency Control. IEEE Trans. on Software Eng., Vol. SE-5, No. 3, 1979, pp. 203-216

Bj75 Bjork, L.A., Jr.: Generalized Audit Trail Requirements and Concepts for Data Base Applications. IBM Systems Journal, Vol. 14, No. 3, 1975, pp. 229-245

Bu77 Bubenko, J.A.: The Temporal Dimension in Information Processing. Architecture and Models in Database Management, G.M.

Nijssen, Ed., North Holland, 1977, pp. 93-118

Ch76 Chamberlin, D.D. et al.: SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control. IBM Journal of Research and Development, Nov. 1976, pp. 560-575

Ch82 Chan, A. et al.: The Implementation of an Integrated Concurrency Control and Recovery Scheme. Proc. SIGMOD 82, Orlando, Florida, June 1982, pp. 184-191

CW83 Clifford, J., Warren, D.S.: Formal Semantics of Time in Databases. ACM TODS, Vol. 8, No. 2, 1983, pp. 214-254

Da82 Dadam, P.: Concurrency Control and Recovery in Distributed Databases: Fundamentals and Concepts. Ph.D. thesis, University of Hagen, Dept. of Mathematics and Computer Science, 1982

DPS83 Dadam, P., Pistor, P., Schek, H.-J.: A Predicate Oriented Locking Approach for Integrated Information Systems. Proc. IFIP 83, Paris, France, Sept. 1983, pp. 763-768

El82 Elhardt, K.: Das Datenbank-Cache: Entwurfsprinzipien, Algorithmen, Eigenschaften. Technical University Munich, Dept. of Mathematics and Computer Science, Techn. Rep. No. TUM-I8208, Mai 1982

Gr78 Gray, J.N.: Notes on Database Operating Systems. Lecture Notes in Computer Science, Vol. 60, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1978, pp. 393-481

HR79 Haerder, Th., Reuter, A.: A Systematic Framework for the Description of Transaction-Oriented Logging and Recovery Schemes. TH Darmstadt, Fachbereich Informatik, DVI 79-4

HL82 Haskin, R.L.; Lorie, R.A.: On Extending the Functions of a Relational Database System. Proc. SIGMOD 82, Orlando, June 1982, pp. 207-212

Ja84 Jaeschke, G.: Recursive Algebra for Relations With Relation Valued Attributes. Heidelberg Scientific Center, Techn. Rep. 84.01.003, 1984

JS82 Jaeschke, G., Schek, H.-J.: Remarks on the Algebra of Non First Normal Form Relations. Proc. ACM SIGACT-SIGMOD Symp. on Principles of Data Base Systems, Los Angeles, March 1982, pp. 124-138

Ki83 Kinzinger, H.: Erweiterung einer Datenbank-Anfragesprache zur Unterstuetzung des Versionenkonzepts. Sprachen fuer Datenbanken, Informatik-Fachberichte 72, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1983, pp. 96-112 (in German)

Kl81 Klopprogge, M.R.: TERM: An Approach to Include the Time Dimension in the Entity-Relationship Model. Proc. Second Int. Conf. on Entity-Relationship Approach, 1981, pp. 466-512

KL83 Klopprogge, M.R., Lockemann, P.C.: Modelling Information Preserving Databases: Concequences of the Concept of Time. Proc. VLDB 1983, Florence, Italy, Oct./Nov. 1983, pp. 399-416

KR81 Kung, H.T., Robinson, J.T.: On Optimistic Methods for Concurrency Control. ACM TODS, Vol. 6, No. 2, 1981, pp. 213-226

La73 Langefors, B.: Theoretical Analysis of Information Systems. Studentlitterur/ Auerbach, Lund, Sweden, 1973

La74 Langefors, B.: Theoretical Aspects of Information Systems for Management. Proc. IFIP 74, Stockholm, Sweden, 1974, pp. 937-945

La82 Lausen, G.: Formal Aspects of Optimistic Concurrency Control in a Multiple Versions Database System. Information Systems, Vol. 8, No. 4, 1983, pp. 291-301

Lo77 Lorie, R.A.: Physical Integrity in a Large Segmented Database. ACM TODS, Vol. 2, No. 1, 1977, pp. 91-104

Lu84 Lum, V. et al.: Designing DBMS Support for the Temporal Dimension. Heidelberg Scientific Center, Techn. Rep. No. 84.03.001, 1984

MS83 Mueller, Th., Steinbauer, D.: Eine Sprachschnittstelle zur Versionenkontrolle in CAM-Datenbanken. Informatik-Fachberichte 72, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1983, pp. 76-95 (in German)

Pa79 Papadimitriou, Ch. H.: The Serializability of Concurrent Database Updates. Journal of the ACM, Vol. 26, No. 4, Oct. 1979, pp. 631-653

PK84 Papadimitriou, Ch.H., Kanellakis, P.C.: On Concurrency Control by Multiple Versions. ACM TODS, Vol. 9, No. 1, 1984, pp. 89-99

Re78 Reed, D.P.: Naming and Synchronization in a Decentralized Computer System. Ph.D. Thesis, M.I.T., Dept. of Electrical Engineering and Computer Science, Sept. 1978

Ro80 Rothnie, J.B., Jr. et al.: Introduction to a System for Distributed Databases (SDD-1). ACM TODS, Vol. 5, No. 1, 1980, pp. 1-17

SP82 Schek, H.-J., Pistor, P.: Data Structures for an Integrated Database Management and Information Retrieval System. Proc. VLDB 82, Mexico City, Sept. 1982, pp. 197-207

Th76 Thomas, R.H.: A Solution to the Update Problem for Multiple Copy Databases Which Uses Distributed Control. Bolt, Beranek and Newman, Inc., Rep. No. 3340, July 1976

Th79 Thomas, R.H.: A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. ACM TODS, Vol. 4, No. 2, 1979, pp. 180-209

We83 Weikum, G.: Entwurfsueberlegungen fuer einen Versionen-Manager zur Realisierung eines temporalen Datenbanksystems. Techn. Hochschule Darmstadt (W. Germany), Fachbereich Informatik, Bericht DVSI-1983-A1 (in German)

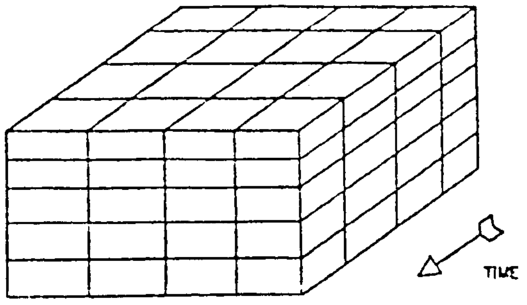

FIGURE 1: Tuples in Table

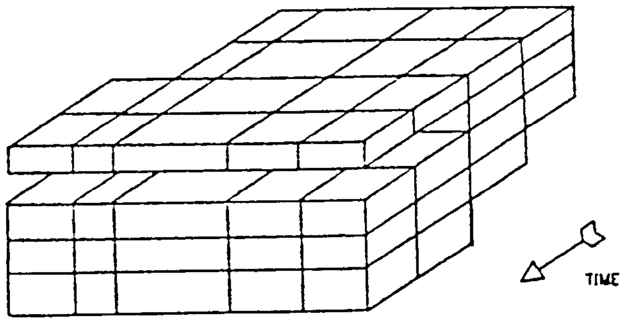

FIGURE 2: Tuples in Table

EMP:

DeptNo	ManNo	Name	Salary	FROM-TIME	TO-TIME
1345	12239	Jones	3,000	800123	810312
1345	12239	Jones	3,500	810313	820220
1345	12239	Jones	4,200	820221	830312
1345	12239	Jones	5,000	830313	∞
1345	17887	Miller	3,500	810210	820211
1345	17887	Miller	4,500	820212	830305
1345	17887	Miller	5,500	830306	∞
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

FIGURE 3: Relation with Additional Time Attributes

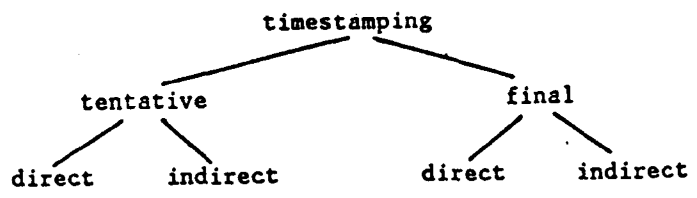

FIGURE 4: Possibilities of Assigning Timestamps to Objects

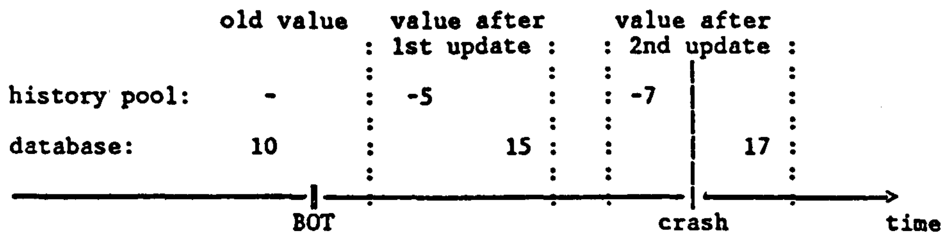

FIGURE 5: Delta Creation in the Case of Repeated Updates