Designing DBMS Support for the Temporal Dimension

V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor,
G. Walch, H. Werner, J. Woodfill

I. INTRODUCTION

As described in the literature, (CO70, CO71), data base information is generally perceived to be time varying. However, in systems in use today, the approach is invariably 'current view'. That is, the information in the data base represents a snapshot of it at an unspecified instant of time and the information is deemed to be the 'current' data. While such an approach is satisfactory in many applications, recent studies in new data base applications have revealed that it is not satisfactory in many cases. New functions for new requirements are needed. Among these new requirements is the need to have temporal, or time domain, support. This paper deals specifically with this particular subject. We shall, without distinction, refer to the support of the time domain as temporal support, history data support, or simply time support. We shall refer to queries with reference to the time domain as time, temporal or history queries.

That the data bases are time or temporally related has been recognized for quite a long time (BJ75). In some cases, as in BU77, LA75, LA76, and SC82, this parameter is brought out explicitly in their models. In most cases, this effect of time is completely hidden in the 'current view' model. Under this circumstance, much of the information about events that have occurred is not available. As a result, certain applications are not possible.

The need for a DBMS with function to support history information can be seen in many applications. For example, in the storage of legislation, historical information is just as important as current data. When a new version of a law is created, the old versions cannot be discarded. Another example lies in office automation applications. In this environment, historical information is of paramount importance. For example, suppose that a document requires two signatures. Information to indicate which one of the two signs the document first can be crucial, if both persons are authorized to change the documents. In case of dispute, even the knowledge of the actions in sequence is not sufficient, unless the versions are stored along with the actions. Still other applications can be found from actuarial and statistical applications. If one were to ask what is the trend or the behavior from a certain perspective, it would not be possible to answer such a question without history data. Recently a number of articles has been written to show the need to support versions and the temporal aspect (CO79, CL83, KL83, KL81, MU83) and we shall not repeat them here.

It suffices to say that history data can be quite important in many applications. In the past, when necessary, the support for time domains was fulfilled by other means. For example, in some applications where accounting is involved, some of the historical information is kept in the audit trail file. The historical data are then obtained from the audit trail as needed. Or a DBMS is customized to fit the application. In other cases, the applications requiring time support are explicitly structured so that historical information is captured. In these cases, time is inserted as an additional attribute or parameter (e.g. MU83). While these approaches may have been satisfactory in some cases, their success is limited and their use is restrictive. What is used in one application may have to be implemented again in another. The need to have a data base system to support the time domain has been said elsewhere (e.g. CL83). We, too, see that an integrated approach to design time support directly into the data base system is necessary. We shall discuss this point later.

As a result of the discovery that time domain support is needed in many applications, much research work has been started (see reference list). In some cases, the temporal domain is handled in a very restricted manner (KA82). Such an approach makes generalization difficult.

Other studies have concentrated on the users' perspective (CL83, KI83, KL83, MU83). Here the researchers attempt to see what kind of logic, interface, language or query facility would be needed to support the different usages involving the time domain. Little serious effort has been expended to implement a system with general function for time support as well as the normal data base functions. In fact, it has been said in both Bubenko (BU77) and Clifford (CL83) that such a system may be impractical. While there was not any reason stated to indicate why it is impractical, we assume that what is meant is that such a system may cost too much in storage and performance. We agree that the additional requirement to support history data does add complexity to the system. However, we do not agree that it is impractical. As we shall see later, such a system can be designed to have efficient support for both the 'current view', as in the 'current view' systems, and history data processing as well.

In the following sections, we shall first present a simple perspective and a model from which time support can be built. For this paper, time support will be restricted to the relational model. We shall indicate the difficulty of constructing general time support on top of an existing data base system. We shall show that time support has different interpretations and that using only one time parameter would not be sufficient. Next we shall present a basic design of a system that can efficiently support the normal current-view data base functions and in addition the general time domain support. In this design, not only history data is kept in the data bases for query processing, the historical access paths are preserved as well. We shall then show that there are alternatives to designing such a system and more work is needed to see which alternative may be best. In addition, we shall present in the Section IV the handling of corrections for errors, future time, need for additional new structure, and shall briefly mention how such a system can be used dynamically to allow data structure changes. We shall try to indicate, whenever appropriate, where alternatives are being studied, and where a decision has been made in our implementation of such a system. For more design details, refer to DLW84.

II. THE MODEL AND REALIZATION ANALYSIS

II.1. Basic Model for Time Support

As mentioned earlier, in this paper, we shall restrict ourselves to the relational model, with data organized in tabular (flat) form. The data entries in the tables will be referred to as tuples, and each tuple will be made up of attribute or field values. We shall assume that the normal understanding of the relational model applies.

When the time aspect of the information is to be explicitly stored with the data, a table with entries can be viewed (BJ75) as a 3-dimensional structure (Fig. 1). In this figure, it can be seen that the approach in today's data base system is to store a vertical slice of the data at a given time, which in general is the most current slice. To be able to handle history queries, we need to store all our data from the time of their creation, including all the intermediate values generated by changes. Of course, to be able to talk about 'time', it is necessary that every system must have a clock where the time values can be obtained. We shall assume that such clocks are available and that they will always give sufficiently exact physical time values.

Because time is continuous, the stored data never truly reflect the exact picture in Figure 1. What is defined in a data base management system is a set of 'functions' which together with the stored data will reflect this model. When data remains unchanged after creation or modification, it can logically be assumed to have only the most recent values assigned to it. This is the normal way to interpret data values in practically all the data base systems. For example, in the current view model, the time at which the snapshot of the data has been taken is never defined. The time the view of the data is valid is assumed to be the time the query is being processed.

A more realistic model of table history is given in figure 2. This representation captures individual upates, insertions, deletions, and structural changes, all occuring randomly in time, and its interpretation is also consistent with other work in this same subject area (e.g. SE80).

Further, in a relational table with no time support, data tuples can be uniquely identified by specifying the key of the tuple. In the model with time included, one must include the time aspect with the tuple key in order to uniquely identify the tuple being addressed. Naturally, when the time value is not given explicitly, one would assume the time of interest to be the time when the query is being processed.

Because much work has been done in processing tables without time support, one would want to see if the 3-dimensional diagram can be converted into a 2-dimensional table. Figure 3 is an attempt for such an approach. It permits us to implement the time-support model directly in tables since we store the creation or modification times as user data. Deletion, in this model means that the data is modified and the values from the time on will be nonexistent.

While the model to include the information of the time domain with data is simple, and it seems that a direct transfer from the model to implementation is possible, additional thought reveals that the task of implementation is rather complex when the various issues are included in the design consideration. We shall now discuss some of these issues.

It is obvious that Figure 3 can be implemented directly on any relational data base systems. The additional attribute, TIME, will be used to store the real clock time values for each tuple. When a tuple is created, the time value of the clock at that instant will be stored.

However, it is not sufficient to have only one attribute for time. While it is adequate to support creation and update activities on data bases, it is not adequate to support deletion. To support deletion, one can assign an area in the header space to store such information or one can create a second attribute so that the two time attributes represent DATA-VALID-TIME-FROM and DATA-VALID-TIME-TO as in MU83 or some other equivalent alternatives. Assuming that we have now adopted this approach, have we solved all our problems in dealing with time? The answer is definitely "no".

II.2. Logical Time and Physical Time

In all publications encountered, where the time domain has been discussed, it was assumed that time is a single parameter. That is, one parameter will be used to represent all the meanings of time. A closer look into this aspect suggests that the time so discussed is generally meant to be the physical time, i.e. the time on the non-stop running clock. While physical time is necessary to be the reference point, we think that there is another aspect of time that must be considered. That is, the time associated with the user's application perspective.

As an example, consider the relation EMP (EMP#, SAL). Let us say that employee #5 had salary $1000 on 8/1/83. Suppose now his salary has been changed to $1500 on 12/1/83. The system will record that fact. However, suppose that on 12/1/83, his salary is to be changed to $1500 but retroactive to 8/1/83. There is no way in a simple meaning of time to reflect this fact, because if we actually change the history salary to reflect $1500 as of 8/1/83, we will have lost some information. If we don't, we will have no way to know that the salary is retroactive.

The problem is that one needs two time parameters to describe the situation. For example, if a query is 'What was the employee's salary on 9/1/83?', the answer can be $1000 or $1500, and each of them is valid. If we give the answer $1000, we mean that, on 9/1/83, his salary has not been changed and it was $1000. On the other hand, if we give the answer $1500, we mean to say that his salary has been changed retroactively to $1500, and if we look at the salary after 12/1/83, his effective salary on 9/1/83 is $1500. In order to get a precise answer, we must have a precise query. Here it is necessary to specify another aspect of time, namely, a second reference point in time, before or after the salary change. While this reference point in time has a relationship to the physical time that we used to record the information in our time stamps, it is not the same 'time'. We refer to such time aspect as logical time.

Another case where the single 'time' parameter causes problems can be seen in engineering applications. Engineering applications frequently specify versions of an object, a car, an airplane, etc. These applications also state the date where a version may become effective. The idea is that an object, which is stored as data tuples, will be updated to contain new data at a particular instant in time.

However, when this effective time refers back to the past, and an event actually has occurred, then it is history data and that time is 'cast in concrete'. But when the time refers to the future, or refers to some event that has not occurred, (e.g. a part scheduled to occur but for some reason has not happened), that time, i.e. the effective time, is 'soft' and it is not the same as the time from a clock. The change in this 'time' must be permitted as it is not really time in the sense of history or real time. While future times are related to the real time, the real time is only the reference point. In MU83, the authors attempt to use one time for both purposes. They permit 'time' to be changed if it is in the future but disallow changes in the past. This is found to be unsatisfactory. In their approach, it is not possible to store the information about the effective time of an object and the time when the object is created or when the effective time has been modified.

One may ask the question if this is necessary. The answer is positive. Frequently actions are causal and consequential. The fact that action 1 exists before action 2 or vice versa is of most importance. We must have a mechanism to record the physical time as a reference point, allowing no one to temper it, and other time aspects that users have some control over them.

To distinguish these different 'time' notions, we propose to use the physical time to record all data base actions and let the users define another one or more time parameters explicitly in the data base. These 'time' parameters are referred to as logical time based on the real physical time as a reference.

Thus, for example, the time that an engineering release becomes effective can be named as EFFECTIVE-TIME and is an explicit attribute. This time can be changed by the user at will. However, each data base action will be recorded. Similarly, to indicate that the employee salary can become retroactive, an additional attribute SAL-EFFECTIVE-TIME can be included in the table. One can then answer the kind of query illustrated earlier, provided the query is specified completely.

Because all logical time aspects must use physical time as a reference, and because we think that all the logical time aspects can be supported by constructing additional functions in the system, when the physical time is always captured, our discussion in this paper is

limited to the design and implementation aspect of the physical time, except for sections IV.1 and IV.2.

II.3. Appending Attributes to Tables Insufficient

In SE80, it was argued that, from the logic point of view, the approach of appending the time attribute to a relational table to support the time domain is not desirable. Here we shall show that, from an operational point of view, such an approach is ineffective as well. We shall show that time support must be designed directly into the system in order to be effective.

With the method of appending time attributes to tables, all tuples belonging to the same table will be treated as equivalent, regardless of the time aspect. The history of all the different actions, delete, update, insert, or create, will be treated as equals and each action will add one or more new tuples to the table. This will cause the table to grow very fast, unless the table is static or semi-static. The result is that we will have performance and storage problems. Perhaps, these were the reasons that prompted Bubenko (BU77) to say that "...we cannot - for practical reasons - realize (implement) information models with the full temporal generality ..." and Clifford (CL83) to say it is "prohibitively costly" to have a "direct implementation".

Adding further complication to the problem is that historical access paths need to be maintained. It is a general practice in data bases to create additional access paths other than the sequential access to the tuples in the tables for efficient processing. The most common technique is to create indexes on attributes according to various strategies. These index access paths should be maintained to have satisfactory performance.

If a data base system maintains indexing only for current data, we would be frequently forced to process history data by a sequential scan of all the data in the table. This scan is caused when data values are changed, and the old values will not be visible any more. For example, in a relation EMP (EMP#, SAL), a question like "Who had a salary of $20,000 in 1977?" will cause the system to sequentially scan all the current and deleted tuples and go into the history to 1977 to find the answer, even when indexing on SAL is implemented. To avoid such expensive processing, we must extend our concept to include history information for the indexes. By appending time attributes and by providing indexes on the SAL and TIME attributes at the same time, one can obtain some of the effects of maintaining historical access paths.

In addition, as a result of business or technical need, there is a good chance that the access paths and table structures may have to be modified during the life time of the data base. For example, indexes for some attributes may be deleted and others created. Fields or attributes may be added or deleted. (In System R (CH76), even without history data, some actions to modify the table attributes are allowed). Tables themselves may be deleted or created. When such action happens, the DBMS catalog, where such information is kept, must be modified. How to maintain history accessing capability as well as current data accessing is by no means simple.

The least to be done to solve the structure change problem is to have functions to manage the catalog so that actions to catalog changes can be accommodated. We could achieve this by adding programs to simulate the necessary catalog functions on top of an existing DB system and by performing catalog history data management. This approach is a very unsatisfactory choice. Not only would the resulting system be unwieldy and inefficient, but it would have other problems as well. Since the new functions are viewed as applications to the original DB system, many of the functions in the original DB system, (e.g. consistency, concurrency control, transaction management), will not likely work properly with the added functions.

It is our contention that the only alternative is to design the function of history data processing into the DB system directly. We must integrate the handling of history data into the processing of current data.

III. SYSTEM DESIGN TO SUPPORT TEMPORAL PROCESSING

III.1. Table Structure with History Data

Figure 2 is simple and conceptually lends itself to easy implementation. In this section, we shall present a basic design to implement that model. We wish to have a design that can handle time support efficiently but with as little semantics as possible for easy and straightforward implementation.

Instead of building one table with additional time attributes to contain all the tuples and changes since their creation, we shall have a table that contains only the current tuples as is done in the current view systems. However, to capture the time aspect, we will have the time stamp stored with each tuple. This time stamp does not behave in the same way as in the case where the temporal domain is supported by adding the TIME attributes to the tables. There the TIME parameters will be just application data and does not have any system protection as provided for system data. Here the time stamps will be protected and are normally transparent to users. Included with each tuple will be a pointer that points to the first history tuple when there is one. Conceptually all history tuples have the same structure as the current tuples.

The method for creating and storing history data works as follows: All history information

belonging to one tuple is chained in reverse time order. The beginning of the chain is always in the current tuple, except in the case where tuples have been deleted. When that happens, it will be moved to the history chain. In its place, a small delete indicator is kept along with the a pointer to show the beginning of the chain.

With this basic structure, we can process all the current and history data in a table, assuming, of course, programs are written to walk the data and interpret their correct values for a given TIME specification. Thus, for example, in a relation with employee and employee salary information, questions like "Has Jane's salary risen?", "Has Jane ever earned the same salary as Mary?", etc. (CL83) can be answered. In choosing to store data this way, it has been assumed that the recent data is most frequently accessed and the older the data the less likely it will be accessed. Thus the most recent values will be accessed first and oldest accessed last in the chain. This is certainly in keeping with the general notion of the "current view" approach, where only current data is kept and history data is assumed to be useless. A module will be created in the data base system that will know how to "walk" the data and deliver the appropriate tuples according to the time specified.

While this implementation, which has been selected for our system, can process current and history data, it can only do so in sequential accesses. Random or direct accessing is needed. We choose to provide this accessing by indexing as it is normally done in many DBMS's.

III.2. Indexing Support Structure and Strategy

Index accessing is generally implemented with a different access mechanism than that used for data tuples, and general techniques are available for such purposes. We wish to avail ourselves of this knowledge and provide a simple design that can process history information with little degradation on the processing of current data. The method proposed in this paper has this property, although other alternatives exist.

Before we go into discussion of our proposed method, we would like to explore briefly the basis that has led us to our choice. A natural notion is to use the same scheme for keeping history information for the index as we do for the tuples in the tables. That is, we first build an index tree and treat each index block as if it is a tuple. We then build a history chain for these blocks so that one can always get the history index tree by following the same principle as done for the tuples.

Unfortunately, because index tree blocks split and recombine, such a scheme leads to considerable complexity in the creation and maintenance of the index history information. Moreover, the performance for this approach is also poor, as we must traverse each index block following its history chain. We therefore have been forced to explore other alternatives. In particular, we have been motivated to find a scheme that not only is efficient, but also can use the same structures that we must provide for other purposes in this system. This has led us to the method that we will now discuss.

Figure 4 is a schematic diagram that illustrates our approach. We organize two trees for index accessing, the current index tree and the history index tree. Each of them is just an ordinary tree, for example a B-tree, B*-tree, etc. We also have another area where pointer lists (lists of pointers to the tuples in the tables) are stored. The internal nodes of the two trees are just the normal tree nodes. The leaf node in either one of the trees has only (index value, pointer) pairs as shown in the structure of the leaf block. The pointers in these leaf nodes point to the various pointer lists. Each pointer list and its history chain will correspond to one and only one index value from one of the two trees. Moreover, each ponter list will be treated as a single attribute tuple with variable field length. The maintenance of the history data chains is done by "infield" operations, the details of which are described in DLW84.

The index values in the current index tree contain all the values of that attribute in the current tuples; the values in the history index tree contain basically those from the tuples that existed in the past but no longer in the current tuples. Neither tree contains time information. Index search will always start in the current tree. For current data query, when a desired value is not found the search process stops. For queries involving history data, the search continues in the history tree. If nothing is found there, then that index value never existed.

Addition of data tuples to the data base, when these tuples create no new index values in the current tree, causes modification only to the pointer lists, but not the index trees. Deletion of data tuples, when no index value is deleted from the current tree, again requires maintenance only in the pointer lists. The method of handling the changes to the pointer lists will be similar to that done in the data tuples of the tables. That is, history data will be created, and the history pointer in the current pointer list will point to the history pointer list (Fig. 5).

In the case where a new tuple causes a new index value to be added to the current tree, then the current tree must be modified and a new pointer list created (Fig. 6). This operation is very simple, as it is only an insertion of an index value to the current tree. Nothing is to be done in the history tree, although the same value may already exist there. This means that for the same index value, we may have two pointer list chains in the pointer list area, one corresponding to the value in the current tree and the other to the value in the history tree.

When an index value is to be deleted from the current tree, maintenance is needed for both trees and the pointer list area. When the index value to be deleted does not exist in the history tree, that index value will be added with the same pointer as it was in the current tree prior to deletion there. In addition, a delete entry with time stamp and history chain pointer is created to head the history chain. Figure 7 shows the case when the value already exists in the history tree. In this case, we have two pointer lists before the deletion and we must connect the two lists back together so that we will have only one history chain for a given value in the history tree.

The above maintenance strategy has been selected to minimize insertion time. This strategy has the possibility of producing overlapping values in the two trees. Thus, the search for history data only through the current index tree may not produce the desired history information, since we may have two history information chains, one from each tree. When searching for history data in a history chain is done via the current index tree, and when the time of the history data at the end of the chain is not as old as the time specified in the query, search must be continued via the history tree to determine if there is another history chain.

III.3. Alternative Strategies

The scheme described so far has been selected for our system; it allows access to all current data with little penalty, although accessing the history data may require an additional tree search. This is consistent with our goal of designing a method that penalizes little, if at all, the processing of the current data, but puts the added cost in history data processing.

Depending on the different situations, various alternatives to the design for history access path maintenance can be developed. The alternatives are many and it is not possible to discuss all of them. However, it is worthwhile to point out a couple here.

Since history data is cumulative, the history tree can be very large and performance degradation may occur. In this case, to reduce the number of levels in the history tree, we can create a number of smaller history trees. For example, we can have the first n leaf nodes in the current tree correspond to history tree #1, the next n history tree #2, etc. This information can be stored directly in the current tree leaf nodes so that the appropriate history tree will be accessed. When a leaf in the current tree splits, the new leaves will inherit the same history tree as the predecessor leaf.

In fact, we can create as many history trees as the number of leaves in the current tree before any update or delete activity. If we do this, the history trees will be very small. In this case, we may want to have an alternate strategy for maintaining the two trees and the pointer lists. We can make the values in the current tree and the history trees non-overlapping. This can be done by deleting the value from a history tree whenever that value is to be created in the current tree. Naturally the pointer list must be connected correctly at that time so that the history data can become complete.

This alternative slightly penalizes the activity of inserting new values to the current tree. However, since the history tree is small, the added cost is very little. Further, when the attribute values to which the index corresponds are non-unique, creation of new index values is likely to be infrequent. The avoidance of having to search the history tree every time, when data is not found through the current tree accessing, may make this trade-off appealing. However, since tree blocks may also recombine, the history trees may be forced to recombine as well. As a result, this scheme may be advantageous only in special cases.

In the case of unique-valued attributes, the pointer lists would contain just one pointer value. Thus, one might consider to put directly this pointer into the corresponding leaf node of the current index tree. However, the pointer list table -&rbl.together with the history index tree&rbl.- remain necessary for maintaining history information. Moreover, this alternative approach assumes that the index entries in the current history tree are created at the same time as the associated tuples, unless additional facilities are included for maintaining the current tree.

III.4. Space vs Access Time Trade-offs

In the basic design described, we have indicated that history data is stored and connected by pointers. This makes it time consuming to search for history data when the chain is long and the required data is at the end of the chain. To avoid this costly search, we can extract the time stamps in each history tuple and put them into a list containing only time_stamp-pointer pairs where the pointers point to the corresponding history tuples. Instead of pointing to the history data that immediately precedes the current data, the pointer in the current data will point to this time list. Figure 8 shows this new strategy.

While it seems that this alternative is always faster to access history data, this is not exactly true. To gain better access to old history data, we have inserted a time list. This means that an additional access is necessary for every access to history data, regardless how recent. Thus, if the average history chain search is not more than three, it is better not to use this alternative.

In the case that history data activity is great, the time list may be very large. Then one may even want to organize this history list in a tree to avoid a linear search. However, since this list can be, and should be, ordered, even a

long list can be processed quite efficiently and such precautions may not be needed.

Storing complete history information in each history tuple, a prerequisite of the method shown in Figure 8, is likely to require a huge amount of storage space. In many cases, this may not even be possible. Since current information is stored in its entirety, it is sufficient to store only the changes (called deltas) in the history data. This scheme is particularly suitable for the basic chaining method in the handling of history data. It is, by the way, in full accordance with Bjork's proposal (BJ75) for recording audit trail data.

To store only the changes in the history data, we need an efficient scheme for generating the deltas. Since these deltas can come in arbitrary form, a general and optimal algorithm can be quite complex. An algorithm to compare before and after images and store the bit patterns that allow regeneration would be too slow, particularly when the data string is long.

A compromise is possible. Since insertion and deletion will require full images in any case, we only need to look at the updates. Generally updates are field or attribute oriented. Hence we need only to store the before-image of the field along with an indication to show which field has been updated. This may make the deltas sufficiently small and generation and regeneration simple.

Alternatively, one can require the system to specify the position where changes occur, along with the data base action and the complete image of that part that has been changed, each time a delta is to be generated. One or more such image and its starting position can be allowed for one data base action. This information is captured and stored in the deltas for regeneration of the complete image.

In some cases, history data may not be useful at all or only for certain attributes. If so, users of the system can specify which of the attributes should participate in the history data and the system will generate deltas only when those attributes are changed. However, in this case, it is not possible to tell what the values have been for the attributes for which history data is not kept. Users must be aware of this when they retrieve history data.

Note that both approaches can be integrated by storing some information in the header of a history tuple, such that the system can decide whether the piece of data just accessed represents a time-pointer series or a delta. This is advantageous as the users will gain added flexibility. They can start with either scheme and change to the other when needed, after they have observed the effects.

IV. DISCUSSIONS

IV.1. Correction of Error Entries

As discussed in the previous section, time stamps entered in the tuples and the history data are not permitted to change. There are, however, occasions where the users would want to correct certain information contained in the history data. For example, suppose that one has entered data yesterday but found it to be incorrect today. With only the physical time captured, we will have the corrected data and the record of the corrective date base action. That would not be sufficient for the system to make the correct interpretations in the various cases. To be able to make the right interpretation, one must also use the logical time aspect as given before. That is, the user must state the effective time for the correction, which in this example is yesterday.

The system, upon receiving notification that the data base action is correction, will generate a delta which includes the extra information, namely the effective time the data is to be valid. History data, however, will not be changed. The system must then make use of the effective time along with the physical time to provide the correct information. The placement of the correction tuple is important because it affects the system's performance.

Further, if there have been other actions since the time the incorrect data has been entered, (i.e. the period between the physical time now and the effective time stated in the correction data), corrective data base actions must be provided by the user. This is necessary, for it is difficult to assess the various effects propagated by the erroneous data.

Potentially, each data attribute in a relation can have its own effective time as well. For example, if we have a relation with EMP# and SAL, one can see that the effective date for EMP# and SAL may require different effective time. However, it is possible to break the two effective times into two operations, so that only one effective time is needed.

A detailed algorithm for correction is beyond the scope of this paper and will be reported in another report on this same subject.

IV.2. Handling of Future Time

As we see, the structure of a system and the semantics to process time or history queries are getting more and more complicated. It would be more so when we allow an effective time that is in the future. However, this requirement is quite reasonable in many applications (e.g. engineering). A design and implementation for such a case, however, is rather complex. Complexity occurs because the system must handle future effective time differently than physical time. Data effective in the future should not be at the head of the chain since it is not yet 'current'. However, as the future effective

time becomes current, then the data must be moved to the head of the chain.

A design as shown in Figure 9 may be used for this purpose. Here we create two chains, one for history and another for future. The history chain will grow as data is moved from the future chain to become current. Unfortunately, that is not all. We have to provide correction to the future chain as well as the history.

Instead of designing a system with such complexity, one could also follow an alternative approach. Since a future object is not really the same as a history object in that the object in the future is not yet a reality, one can separate the future instances from the current and the past. Thus, future objects can be stored in different tables and moved, manually or automatically, into the current tables as their effective dates get to become real physical time. However, this approach may require more processing than necessary.

IV.3. Processing Data When Structure Changed

No DBMS known to the authors allows arbitrary on-line changes in its data structure once the data base has been created. Some systems, e.g. System R (CH76), permit the addition of fields or attributes at the end of the current structure. Deletion of attributes is not allowed. To allow arbitrary changes, one must have the history of the catalog data. This can easily be done in the scheme proposed.

To use the same mechanism to store history information in the catalog, the catalog table should be designed just like the data tables. When a change in the data structure of a table occurs, history information will be created for the corresponding tuple in the catalog table. Tuples in the table where the data structure has been modified need not be changed at all. To interpret the data correctly, one must create the appropriate data structure from the catalog history information corresponding to the time that data was created. The implication is that several versions of the data structures may be needed even when a query requests only the current data. This happens because the tuples in the table may have been created at different times with different data structures.

IV.4. Need for Additional Access Structure

The design presented is sufficient to support the normal data base applications that process history data as well. However, we may also be interested to support queries which ask whether and when the stored history information contains certain events. For example, suppose the query states, 'Give all the employees who have ever received a 10% or more salary increase'. Without additional access path support, to answer such a query would require a sequential search of all the related tables, both current and history data.

One possible solution is to create a special index as illustrated in the following: For example, suppose that the events that we are interested in are the differences in the salary changes. Each time a new salary is entered, the difference (indexing deltas) between the new salary and the old salary is computed. (In the case where the percentage, but not the absolute value, is the event of interest, as in the example cited, the percentage (again the indexing deltas) of increase will be computed at the time the data change occurs.) The indexing deltas together with the event time and the associated tuple-ID's (TID) are collected as shown in Figure 10 and treated as an index file.

Sometimes, it may not be practical to have the indexing deltas strictly equal to the indexing values. For example, in the case of salary, it may be too fine a division in storing the indexing delta values of percentages as actually occurred. We may not have any two indexing deltas that are exactly the same and the tree will be too large for our purpose. We can group them together to form interval indexes. Thus, we may say that all the deltas between 0 and 5% have index value 1, and 5 to 10% index value 2, etc. In this way, we will have the special index behaving just like any secondary index.

Retrieving data is simple. We just access the special index to get to the list of tuples to be searched for tuples satisfying the query. There will not be much deletion since the only time that occurs would be corrections to previously entered but incorrect values. Most of the activities in this index are retrievals and insertions.

IV.5. Reclaiming Space of Evicted Tuples

To avoid update and maintenance activities, the system must have stable pointers. In our system, the pointers are made up of page ID and slot ID concatenated together. This is commonly referred to as the TID. When the original page cannot hold the tuple, its original location will contain the new TID pointing to the location which the tuple now occupies. This is a common strategy (e.g. AS76) and shall not be discussed further.

As all history data is stored in the system, storage space size will grow monotonically. If no data can be purged, eventually one will run out of storage space. Therefore, functions to purge or archive data must be provided. When data is purged, the vacated space must be reclaimed.

Sometimes data must be moved from the locations they occupy for performance reasons. For example, consider the case that many delete actions have occurred. To keep pointers stable, when a tuple is deleted, the location which the tuple occupies at creation time is kept for the deleted tuple. However, only skeleton information connecting this location to the history chain is kept here. After a number of deletions, a page may contain many of the

skeleton data tuples. In either of the above actions, degradation will eventually occur and garbage collection will be needed.

Moving data around in or removing data from this system is not as easy as it seems at first thought. To reclaim or reuse any space, we must be sure that no inconsistency in the interpretation of the data in this location exists. In the case of purging data or removing data for archiving, the history pointer chains are affected and so does the space used to maintain the deleted tuples. One simple way to handle the chain update is to have a mark at the end of the chain to show that some history data has been archived.

Concerning the handling of the vacated space used to maintain the deleted tuples, we can do that in two different ways. One way is to clean up the access paths that may lead to this vacated location. This basically means that the indexing information must be updated to reflect the new information. That is, for each tuple removed from the data storage, for whatever reason, that part of the access information must be removed from the index accessing paths. The mechanism for this is not difficult. There are several ways to achieve this result. One can, for example, follow the indexing mechanism to delete the access paths as each tuple is removed from the current tuple storage area. Another one is to do a batch reorganization. In any case, the mechanism for this is not the same as the one for maintaining the indexes. However, it should be noted that, if this approach is to work, no TID's should be given to users. Otherwise, the system cannot guarantee that the reorganized data will not be referenced by an invalid and old TID.

Another way is to have, conceptually, more than one history data chain sharing the same TID (but not at the same time). The history chain that has been removed, virtually occupies the same TID that is now re-used by another tuple, because entries in the history pointer lists, which are not changed, will still point to this location. The new tuple, which uses this location but bears no relationship to the old tuple, will contain a pointer that points to the history chain of the tuple that used to exist at this location. The time stamp information would allow us to resolve any ambiguity. This means that the vacated location cannot be used for simply storing any data, but must be used for data storage where a new time stamp would be created.

As can be seen from the previous discussion, the problem of re-using vacated space is fairly complex. We plan to treat this topic in more detail in another paper.

V. SUMMARY AND CONCLUSION

In this paper, we have discussed the need to have history data support in data base management systems and presented our arguments for the necessity to integrate this goal directly into the design of a data base management system. It is felt that to support historical data access paths, to allow the modification of the data structures in tables, to enhance performance, to minimize storage space, etc. would require such an approach for the resulting system to be effective. This paper presents such a design, as well as some alternatives. The design not only specifies how tuples in tables are to be stored, but includes the mechanisms to maintain the indexes in their historical role. In this manner, all historical access paths are preserved in this system.

While we do not think that all different temporal support requirements can be fulfilled with the proposed design alone, we think that the approach can form the basis needed to implement additional support as demanded by other requirements. We mentioned that, although most people consider that temporal support requires only one additional temporal parameter, there are actually different perspectives from different viewpoints. However, each perspective must be based on the real time as reference.

In addition, storing all history data would cause the storage space to expand very rapidly. To allow us to minimize storage space requirement, methods have been presented to attempt to store only the necessary data. This generally means that trade-offs between space utilization and access time are made. The number of options possible for this goal is large and we have not discussed all of them. However, it is believed that the approach taken in this paper is a reasonable one and definitely is a good first step in the right direction.

Contrary to some opinions that support for history data is impractical, we believe that our design is efficient: a system constructed in the manner stated in the paper can process current data almost as efficiently as a system with only 'current view' data processing support. The price that we pay for supporting history data processing lies mainly in the storage cost when history data is not required, additional processing cost when history data is actually searched and the overhead of creating the history information. This is believed to be a fair solution, consistent with the 'current view' data base management systems.

In many applications, it has been found that the unnormalized or hierarchical data structure can be used to advantage. While we have not discussed in this paper how to design support for such a system, a method based on the same approach as in this paper has been developed. However, due to the added complexity and richer semantics in the unnormalized data structure, some additional mechanisms are needed.

Currently the proposed design is being implemented in a data base management system. Because the data base system supports not only the relational model but the hierarchical or unnormalized data model as well, the design has actually been expanded to include additional mechanisms. In spite of this added complexity,

we believe that the design can perform as effectively and efficiently as expected.

There remain a number of problems that have been partly solved or not yet solved. Among these problems are the system's support for logical times and how to migrate history data to off line systems and be able to recall back to on-line. Another problem lies in the management of data values. If the users do not keep all attributes in the history, then the semantic becomes unclear when history data is needed. The same would be true when attribute type or definition changes in its history. In addition, we have not yet finalized the design for recovery and concurrency control for a system of this kind. It seems that there may be some way of taking advantage of the time stamps in the tuples to provide a better concurrency control strategy. These are but some of the problems remaining to be solved; we expect to come across many more as we continue to build a system with time support.

REFERENCES

AN82:
Anderson, T.L.: Modeling Time at the Conceptual Level. Proc. Second International Conference on Databases, Jerusalem, June, 1982.

AS76:
Astrahan, M.M. et al.: System R: Relational Approach to Database Management, ACM Transactions on Database Systems, Vol. 1., No. 2, June 1976, pp. 97-137.

BJ75:
Bjork, L.A.: Generalized Audit Trail Requirements and Concepts for Database Applications, IBM Systems Journal 14 (1975), pp. 229-245.

BU77:
Bubenko, J.A.: The Temporal Dimension in Information Processing. In: Architecture and Models in Database Management, G.M. Nijssen, Ed., North Holland, 1977, pp. 93-118.

CH76:
Chamberlain, D.D. et al.: SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control, IBM J. Res. Development 20, 6 (Nov. 1976), pp. 560-575.

CL83:
Clifford, J., and D.S. Warren: Formal Semantics for Time in Databases. ACM TODS 8, 2 (June 1983), pp. 214-254.

CO70:
Codd, E.F.: A Relational Model of Data for Large Shared Data Banks, Comm. ACM 13, 6 (June 1983), pp. 377-387.

CO71:
Codd, E.F.: Further Normalization of the Database Relational Model. In: Database Systems, Courant Computer Science Symposia 6, R.Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1971, pp. 65-98.

CO79:
Codd, E.F.: Extending the Database Relational Model to Capture More Meaning, ACM TODS 4, 4 (Dec. 1979), pp. 397-434.

DLW84:
Dadam, P., Lum, V., Werner, H.-D.: Integration of Time Versions into a Relational Database System. IBM Heidelberg Scientific Center, Technical Report, 1984.

FA74:
Falkenberg, E.: Time-Handling in Database Management Systems. CIS-Rep. 07/74, University of Stuttgart 1974.

KA82:
Katz, R.H., and T.J. Lehman: Storage Structures for Versions and Alternatives. Computer Science Technical Report #479, July, 1982, University of Wisconsin-Madison.

KI83:
Kinzinger, H.: Erweiterung einer Datenbank-Anfragesprache zur Unterstuetzung des Versionenkonzepts. In: Sprachen fuer Datenbanken, J.W.Schmidt, Ed., Informatik-Fachberichte 72, Springer, Berlin, Heidelberg, New York, Tokyo, 1983, pp. 96-112.

KL81:
Klopprogge, M.R.: TERM: An Approach to Include the Time Dimension in the Entity-Relationship Model. Proceedings of the Second International Conference on Entity-Relationship Approach, 1981, pp. 466-512.

KL83:
Klopprogge, M.R. and P.C. Lockemann: Modelling Information Preserving Databases: Consequences of the Concept of Time. Proceedings of the 1983 International Conference on Very Large Data Bases, Florence, Italy, Oct. 31-Nov. 2, 1983.

LA73:
Langefors, B.: Theoretical Analysis of Information Systems, Studentlitteratur/Auerbach, Lund, Sweden, 1973.

LA74:
Langefors, B. Theoretical Aspects of Information Systems for Management. IFIP Congress 1974, pp. 937-945.

LA75:
Langefors, B. and Bo. Sundgren: Information System Architecture, Petrocelli/Charter, New York, 1975.

LA76:
Langefors, B. and K. Samuelson: Information and Data in System, Petrocelli/Charter, New York, 1976.

MU83:
Mueller, Th., D. Steinbauer: Eine Sprachschnittstelle zur Versionenkontrolle in CAM-Datenbanken, J.W.Schmidt, Ed., Informatik-Fachberichte 72, Springer, Berlin, Heidelberg, New York, Tokyo, 1983, pp. 76-95.

SE80:
Sernadas, A.: Temporal Aspects of Logical Procedure Definition. Information System 5, 1980. pp. 167-187.

SC82:
Schiel, U.: The Temporal-Hierarchic Data Model (THM), Univ. Stuttgart (W.Gemany), Institut fuer Informatik, Bericht 10/82.

WE83:
Weikum, G.: Entwurfsueberlegungen fuer einen Versionen-Manager zur Realisierung eines temporalen Datenbanksystems, Techn. Hochschule Darmstadt (W.Germany), Fachbereich Informatik, Bericht DVSI-1983-A1.

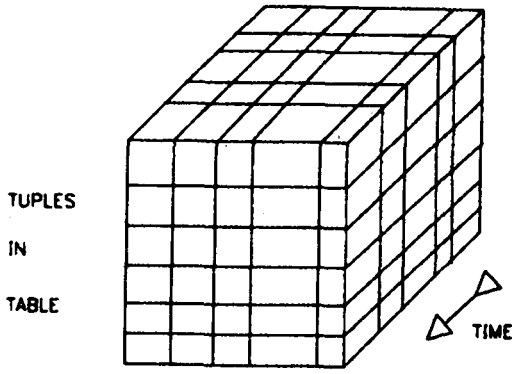

Figure 1

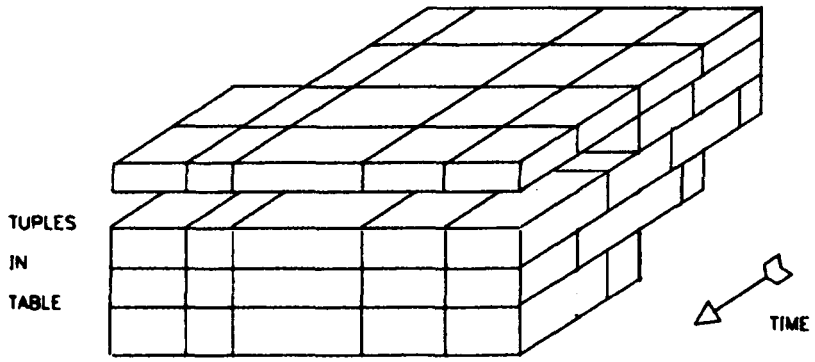

FIGURE2

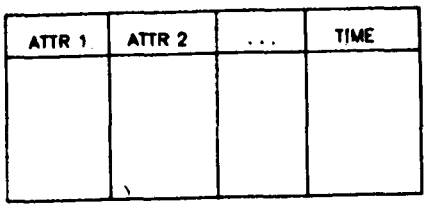

FIGURE 3

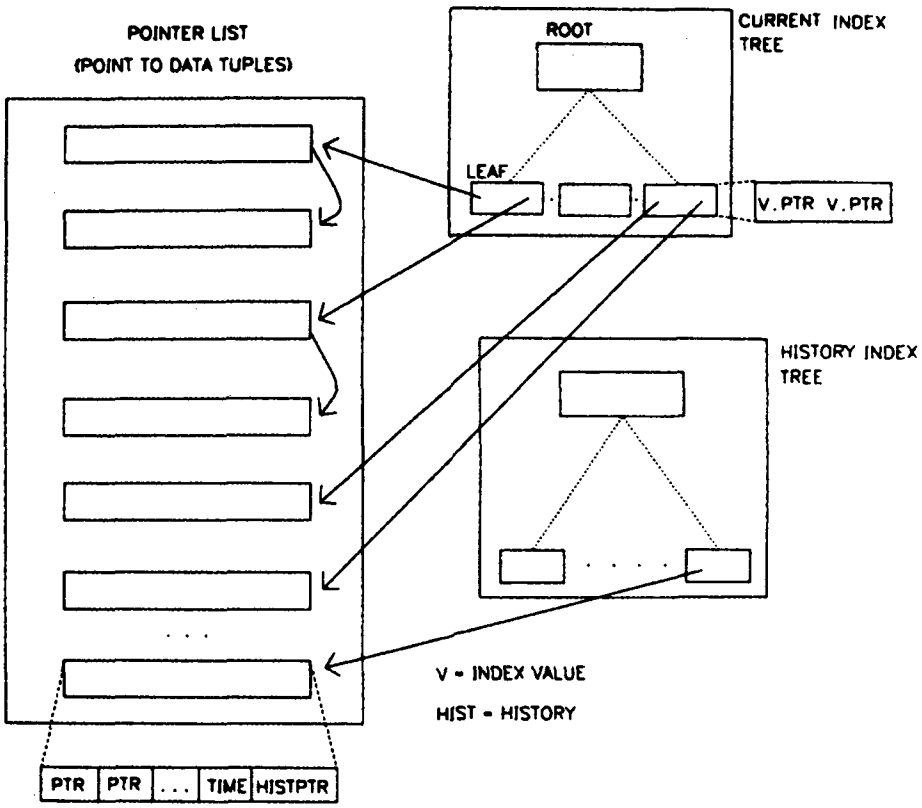

Figure 4

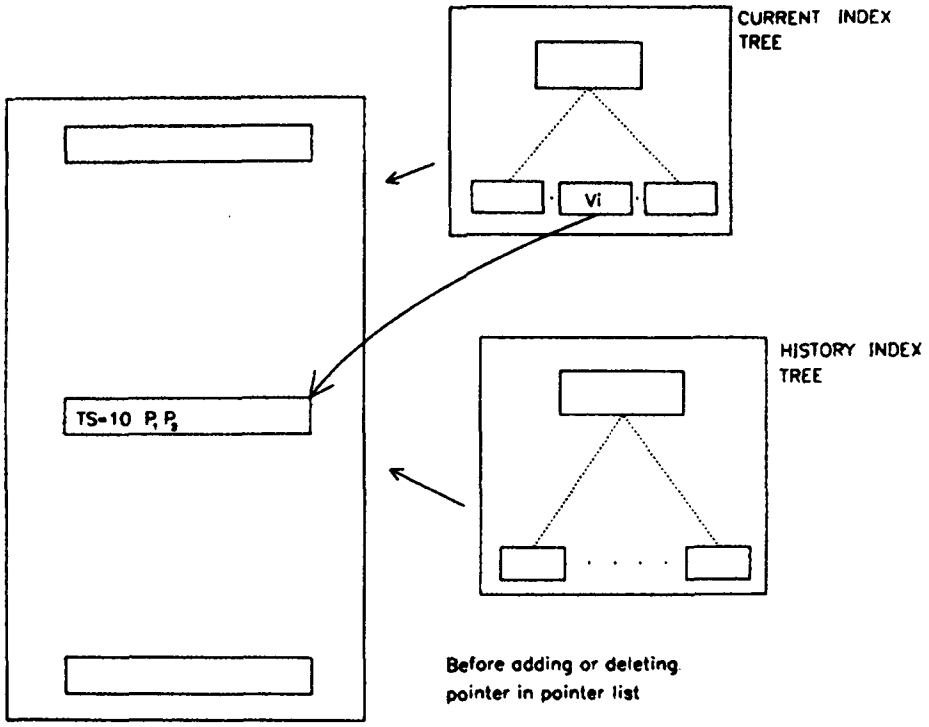

Figure 5(a)

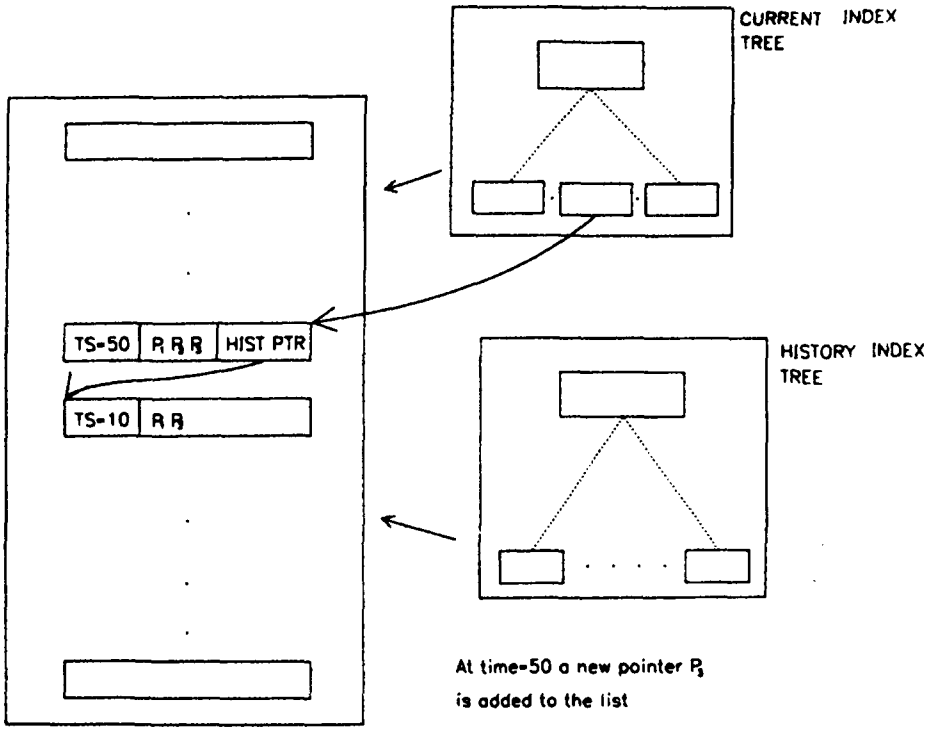

Figure 5(b)

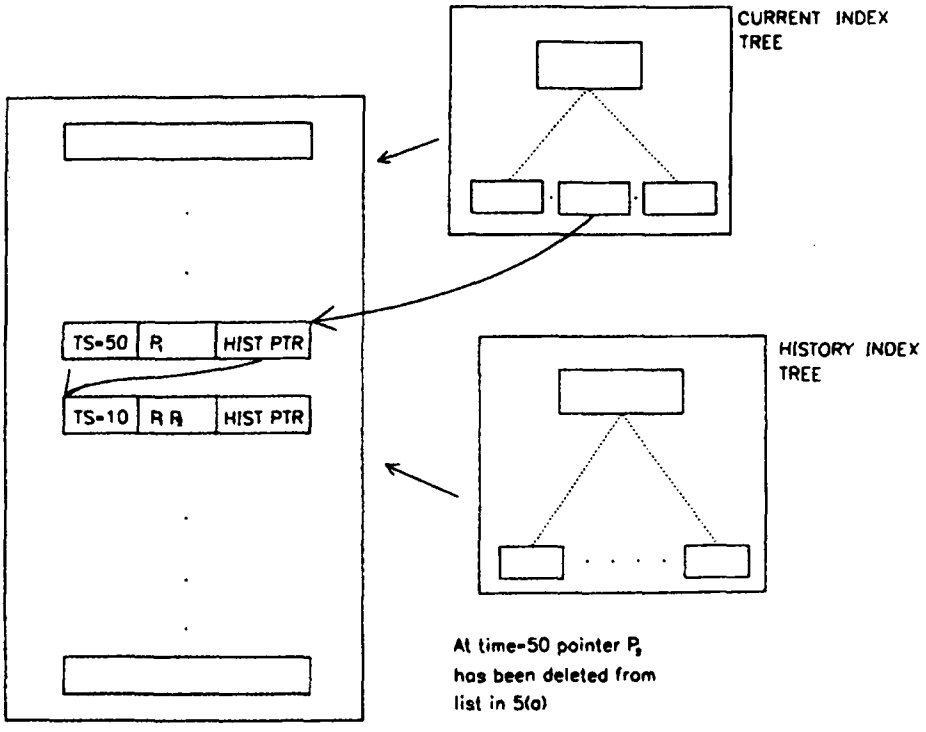

Figure 5(c)

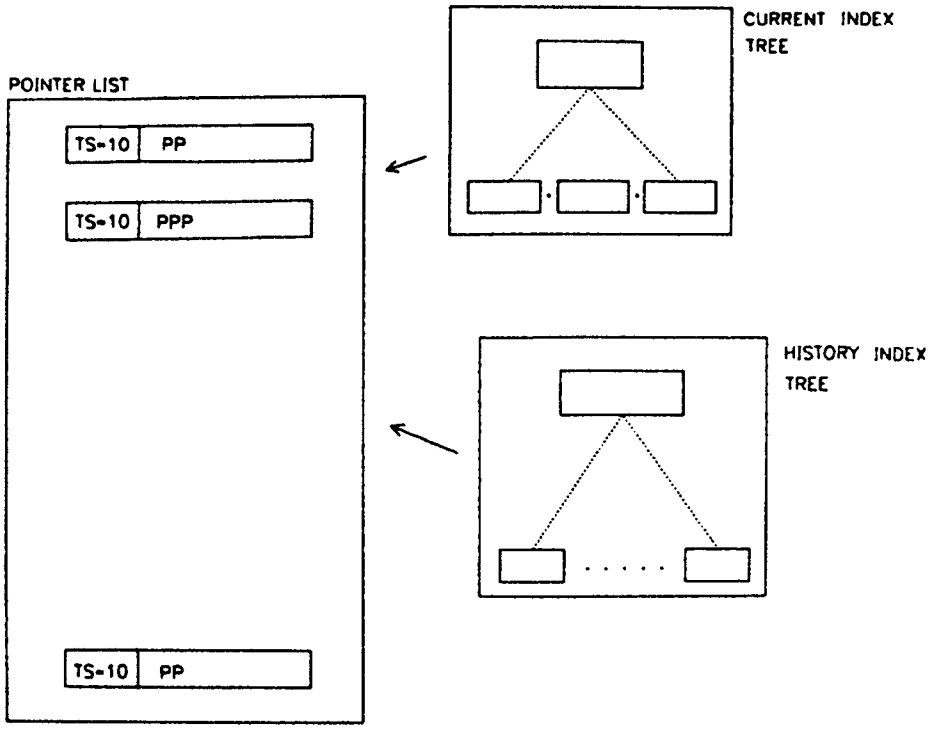

Figure 6(a)

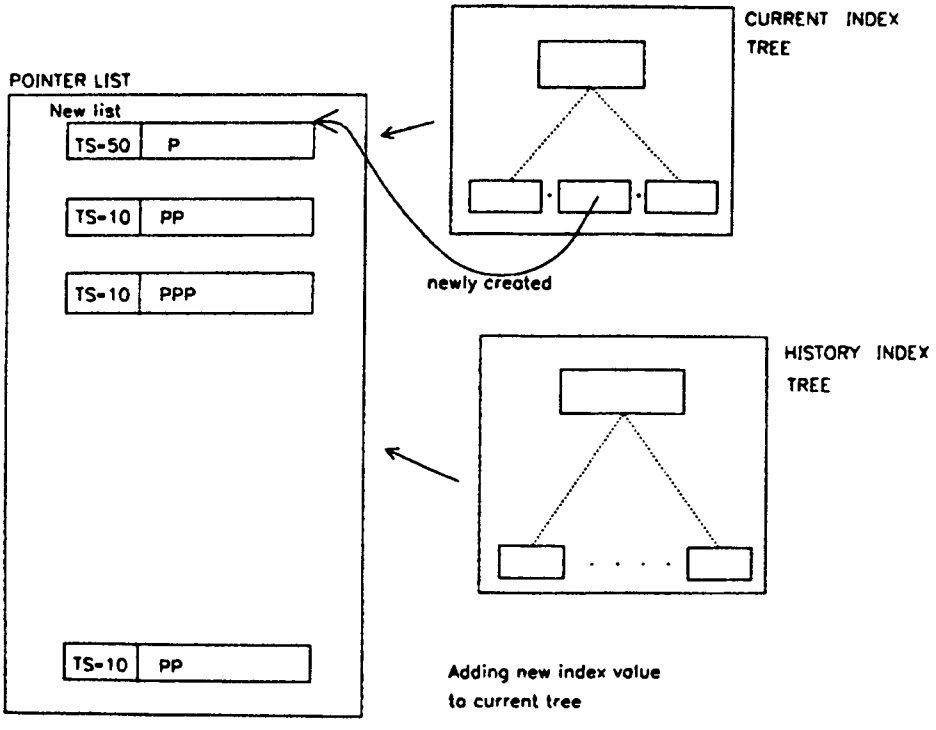

Figure 6(b)

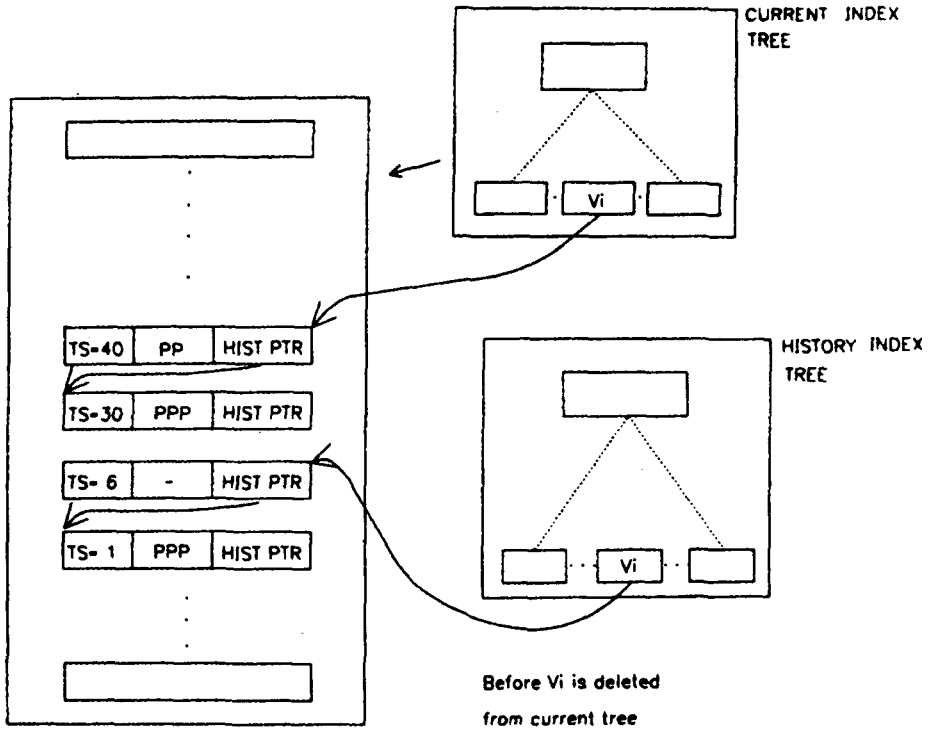

Figure 7(a)

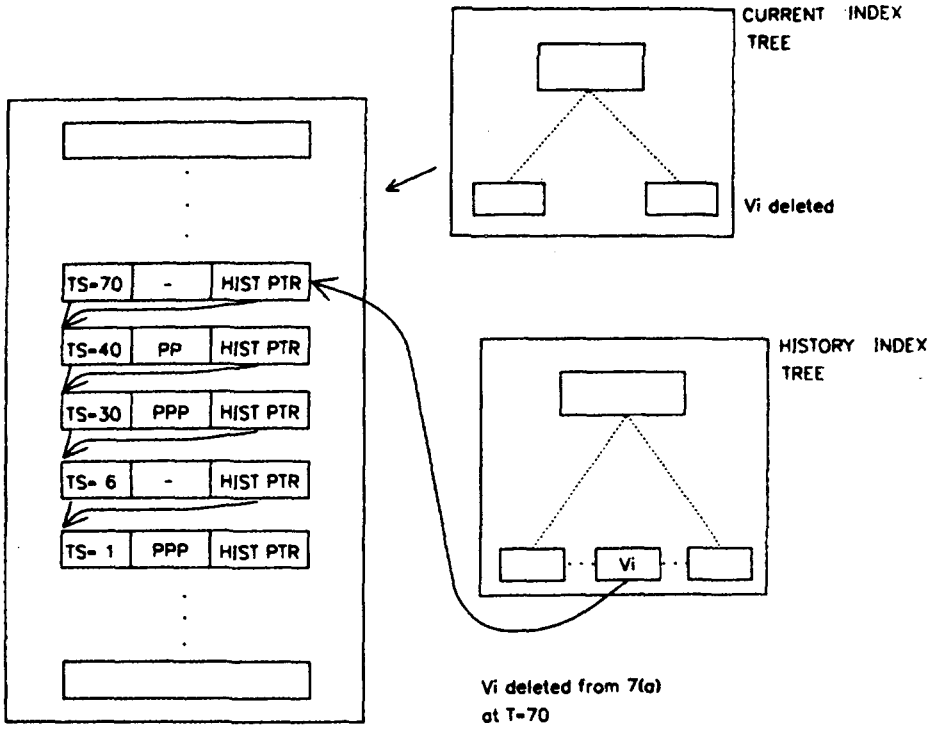

Figure 7(b)

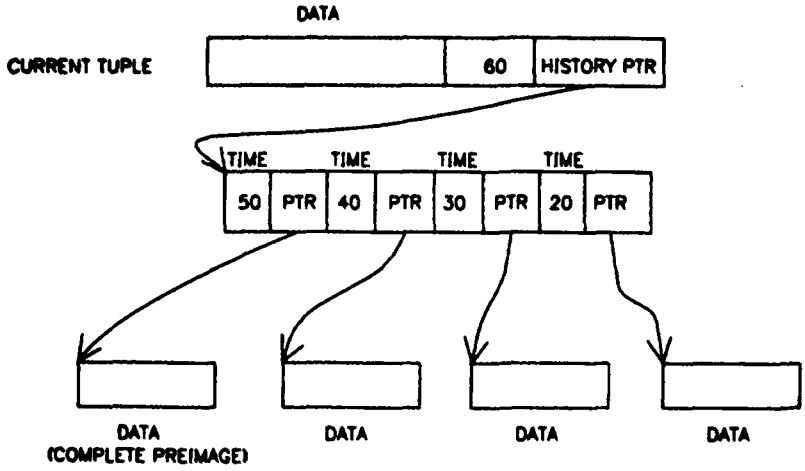

FIGURE 8

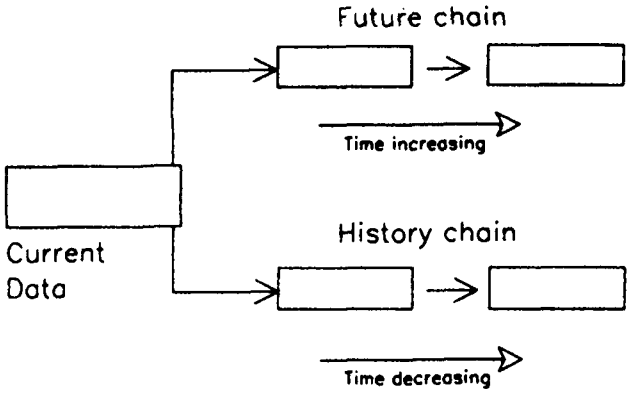

Figure 9

Table structure

A	B	C	D

ΔA index needed

ΔA Table

Δa_1 $(t_{11},TID_{11}),(t_{12},TID_{12}),(t_{13},TID_{13})$ . . .

Δa_2 $(t_{21},TID_{21}),(t_{22},TID_{22}),(t_{23},TID_{23})$ . . .

.
.
.

Figure 10