A new approach to version management for databases

by VINIT VERMA and HUIZHU LU
Oklahoma State University
Stillwater, Oklahoma

ABSTRACT

Following an overview of version management databases, a new approach for implementing these databases by employing the use of persistence in height balanced trees is proposed. Persistence in a tree is obtained by path copying. In classical databases, version management is performed by checkpoints which require $O(n)$ time, and $O(n)$ space. This new methodology maintains historical data in $O(\log n)$ space, and $O(\log n)$ time for a binary search tree. The paper discusses how the concept of persistence is superimposed on B-trees, which are the primary storage structures utilized in present day databases. A search operation on this persistent B-tree of order ‘m’ is $O(\log(k) \log n(\log .80(m-1)))$, where ‘k’ is the number of nodes in an index time stamp tree, ‘n’ is the number of nodes in the B-tree, and the B-tree is assumed to have 80% node utilization.
A New Approach to Version Management for Databases

INTRODUCTION

Human memory essentially has a no deletion mechanism. Memory does exhibit a decay characteristic with time, but people simply do not delete. The concept of deletion was invented to reuse expensive computer storage. With the falling computer storage costs, and new storage technologies (e.g., optical disks), this deletion policy will eventually give way to a non-deletion policy in database applications. All accounting, financial, and legal databases require a non-deletion policy. This policy is usually required by law for good reasons. Once this policy comes into effect, then a version management strategy will need to be imposed on current databases.

Conventional database management systems lack the ability to store and process time dependent data. Without this temporal support the burden comes on the application programmer to build some temporal information management strategy. The resulting applications are inefficient in terms of data storage and processing costs. Most decision support systems require trend analysis and historical queries which are not readily supported by DBMSs of today. Even error corrections have been implemented in DBMSs via audit trails which require space intensive checkpoints to preserve past states.

The initial part of this paper describes the four types of databases which support temporal information processing. The four database types are differentiated by their ability to support these time concepts and their temporal information processing capability. Snapshot, rollback, historical, and temporal databases are the four types of databases which are discussed. In the second section of this paper, the authors propose the use of persistent search trees to implement these version management databases. Persistent search trees primarily utilize path copying for maintaining historical information. The concept of path copying to save historical versions can be carried over to any height balanced tree—height-balanced trees, weight-balanced trees, or B-trees for example. The paper discusses persistence on an HB(1) tree using path copying. The persistence can be obtained without path copying and this is also shown for the same HB(1) tree. By far the most common file structure or access method utilized in databases today is VSAM (virtual storage access method). VSAM is implemented using B-trees. So if this version management can be superimposed on B-trees, we have an appropriate data structure to implement version management databases. The final section of this paper discusses how B-trees can be made persistent.

CLASSIFICATION OF DATABASES WITH TEMPORAL SUPPORT

The four types of version management databases are described below. The relational data model has been used for this discussion. In the mentioned data model the user views the data in the form of tables. The rows in the table form tuples and the columns form the attributes. Inserts, deletes, and updates occur instantaneously on these tables in a conventional system. The table below shows two attributes name and salary, and two tuples (rows).

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jack</td>
<td>$1000</td>
</tr>
<tr>
<td>Tim</td>
<td>$1500</td>
</tr>
</tbody>
</table>

Snapshot Databases

A state, or instance, of a database is its current content, which does not necessarily reflect the current status of the enterprise. Dynamic changes in such an environment are preserved by snapshots. Insert, delete, and update operations on the payroll table occur instantaneously and no historical information is recorded. Such a database is described as a snapshot database. A snapshot database cannot support the following types of temporal queries: (1) trend analysis (statistical information on the number of employees during the past three years), (2) historical query (What was Jack's pay two months ago?). Further, snapshot databases cannot support (1) retroactive change (Tim got a raise of $100 starting last month), or (2) a proactive change (Jerry is joining the company next month) information.

These snapshot databases provide no historical retrievals or updates. The burden to preserve and maintain historical information falls on the application programmer. As the variation of data over time is not related to any application, so the DBMS, rather than the application programmer should take care of this version management.

Rollback Databases

A rollback database stores a sequence of snapshot states indexed by transaction time. Transaction time is defined as the time during which a database update is performed. An update refers to insert, delete, or update database operations. Historical queries can be supported by moving along this
transaction time axis and selecting a particular snapshot state. The operation of selecting a snapshot state is termed rollback, and a database supporting it is called a rollback database.

For example, the relation in Figure 1 has three transactions. Starting from an empty relation, two tuples were added at time A, a third tuple was added at time B, and finally a tuple was deleted at time C. Modifications on a rollback database can only be made to the most recent snapshot state. As is evident from the above transaction, each transaction results in a new snapshot being appended on the transaction time axis. Since snapshot database stores only the most current state of the database, historical queries are not supported. But rollback databases can back up on the transaction time axis and support an historical query. Note, rollback databases support historical queries, not historical updates, that is, insert, delete, update. Thus a rollback database can support the following query on the payroll database: “What was Tim’s salary last month?” The database will roll back on the transaction time axis and display a snapshot of the needed relation. Rollback databases record the history of database transactions. A tuple becomes effective as soon as it is entered into a rollback database. Retroactive and proactive changes cannot be recorded. If it is discovered that Tim got a raise a month earlier than the month that is stored in the database, there is no way to resolve this in a rollback database.

**Historical Databases**

Historical databases record a single historical state per relation. They support valid time—the time during which the relationship being modeled was valid. Since previous states of the database are discarded, historical views are not supported. Historical databases have a resemblance to snapshot databases in that error corrections are not recorded. These types of databases need extra high-level language support to be able to support the complex semantics of valid time. They support the following types of historical queries: “What was Tim’s salary when Jerry joined the company?” The result of the query will be an historical relation which can be used in further historical queries. These databases support arbitrary modifications, whereas rollback databases only allow snapshots to be appended. This historical relation can show that a later transaction has changed the time when a tuple takes effect in the relation. Thus historical DBMSs can represent correct information about the past, whereas rollback databases can back up to an incorrect previous snapshot. Historical relations can be viewed as interval relations, in which a tuple is valid until the next tuple with the same key becomes effective.

**Temporal Databases**

The plus points of the rollback and historical databases are integrated to form temporal databases. These support both transaction time and valid time in the same relation; and both these times can be manipulated by a query. A rollback database views stored tuples, whether valid or not, as of a certain instant in time. An historical database views tuples valid at some instant as of now. Finally, a temporal database views tuples valid at some instant seen as of some other instant and thus can capture both retroactive and proactive changes.

A temporal relation is a sequence of historical states, each a complete historical relation. A rollback on a temporal relation selects a particular historical state on which an historical query is executed. A new historical state is appended to each transaction. Temporal databases are capable of answering the following kinds of queries: According to the state of the database as of December 1985, “What was Tim’s salary when Jack joined the company?” Notice that the query is manipulating both the valid time as well as the transaction time.

The following analogy should prove helpful to an understanding of the temporal classification of databases. A snapshot relation can be compared to the latest payroll stub showing the current salary of the recipient. If the recipient gets a raise, the next stub shows the new salary, but the latest stub gives no information about the recipient’s previous salary. The collection of all payroll stubs forms a rollback relation, a slice of which is a snapshot relation comparable to a payroll stub. No corrections are allowed on past stubs. An historical relation can be compared to a chart containing the salary history of a person until the instant of making the chart. If an error is found in the chart, or a person gets a raise, then a new chart reflecting the change is made. The current chart should always be up to date. A temporal relation is a file of all such payroll charts marked by the date when each was prepared. It is possible to refer to an old chart as it was known at some instant of time. An in-depth discussion on temporal databases has been presented in Snodgrass and Ilsoo.10 Having built the needed background for version management databases, we now consider methods for implementing these databases.

**A NEW APPROACH FOR VERSION MANAGEMENT**

Path copying can be utilized to maintain previous versions of a tree after insertions and deletions.11 Basically, the path in a tree along which an update operation is performed is replicated to maintain the previous version. These trees which maintain previous versions are termed “persistent trees.” A persistent tree differs from an ordinary search tree in that following an insert/delete operation the old version of the tree...
is still accessible. The old version of the tree needs to be maintained after a new version has been created during an update operation. If the entire tree is copied on each update operation, then the processing time is $O(n)$ and the space utilized is $O(n)$. Thomas\textsuperscript{14} addresses the concurrency problem of these multycopied databases. Severance and Lohman\textsuperscript{15} discuss the use of differential files to maintain versions in large databases. In path copying, only those nodes are copied in which a change is made. Thus any node which contains a pointer to a node that is copied, must, itself, be copied. If every node contains pointers only to its children, this means copying one node requires copying the entire path to the node from the root of the tree. Thus, in effect, a set of search trees is created, one per update, having different roots but sharing common subtrees. The processing in a path-copying search tree is $O(\lg(n))$ since only one path in the tree is traversed. The space utilized is also $O(\lg(n))$ since only one path will be copied. If there exists a large number of updates then the roots of these tree versions can reside in an array, or better still, in a search tree. If a search tree is utilized to hold roots for the different tree versions, then a search time of $O(\log(m)\log(n))$ is obtained, where $m$ is the number of updates and $n$ is the number of nodes in the tree. This path copying works on any kind of balanced tree, for example, a height-balanced tree, or weight-balanced tree. In a balanced tree, the balance is maintained by storing certain balance information in each node and rebalancing after an insert/delete operation by performing a series of rotations along the access path. Aho, Hopcroft, and Ullmann\textsuperscript{16} and Knuth\textsuperscript{17} provide a good description of height-balanced trees.

The example below shows how a height-balanced(1) tree can be made persistent using path copying. The keys $CD, AE, BG, QK, RR$ are inserted in that order into an HB(1) tree. The resulting HB(1) tree after balancing is shown in Figure 2. This is the state of the tree at time 0 in Figure 3. The numbers to the left of each node are the time stamps, and the numbers to the right reflect the balance information. At time 1, suppose SS is inserted into the tree. The tree undergoes a rotation as BG becomes critical. The balanced tree with the path $BG \rightarrow QK \rightarrow RR$ copied is shown under the time 1 root. The tree version under time 2 results after TT is inserted into the tree at time 1. Appropriate rotations have been performed to balance the tree. Note the path $QK \rightarrow RR \rightarrow SS$ has been copied from the time 1 tree to result in the time 2 tree. Once a node is copied or a new node is inserted, the appropriate time stamp is placed on the node. A new index tree can be made which holds the pointers to the various time stamp roots. Figure 4 addresses the previous point. If the version 1 tree needs to be accessed, the pointer with key equal to one in the index tree is taken to direct the search in the appropriate tree version. This time-index tree can be made height balanced too.

A search tree can be made persistent without node copying. The nodes are allowed to grow arbitrarily large after updates, that is, each time we need to change a pointer, a new pointer...
is stored in the node along with a time stamp indicating the time of the update. Thus the space requirement is $O(1)$ during an insertion as only one node is inserted and no copies are made. The pointer change also requires $O(1)$ space since only one pointer is inserted. The drawback of the method is the time spent deciding on the correct pointer in a node, as the node has an arbitrary number of left and right pointers. If a binary search on an index time stamp tree is performed, choosing the correct pointer takes $O(\log(m))$, and an access, insert, or delete takes $O(\log(n)\log(m))$, where “$m$” is the number of time stamps and “$n$” is the number of nodes in the tree.

Starting with the tree in Figure 2, a persistent HB(1) tree without node copying is obtained (see Figure 5). The tree in Figure 5 results after the insertion of SS and TT into the tree in Figure 1, at times 1 and 2 respectively. Again pointers to the different version of the tree need to be maintained in an index time stamp tree. The balance tags are overwritten each time, and the arcs reflect the time of creation. To traverse the tree at time 0, the appropriate pointer in the index tree is accessed.

All 0 arcs are accessed once the correct tree version is located. To reveal the time 1 tree, the appropriate index tree pointer is taken again. Then from each node a 1 arc is taken. If a 1 arc doesn’t exist, a 0 arc is taken. The same is done for the time 2 tree. As a tree version is traversed, each visited node needs to be flagged to avoid cycles during the traversals.

Path copying can be carried over to a B-tree also. Figure 6 depicts an order 5 B-tree. At time 0 the B-tree has keys AB,CC,PQ,RR,SS,TT and XY. At time 1, AY is inserted into the B-tree. Notice that the nodes in the path are copied and a new root with time stamp 1 results. At time 2, LM is inserted—again with appropriate node copying. Finally at time 3, KK is inserted resulting in a node getting overfull. The condition is resolved with a 1-2 split. The nodes along the path are again copied, and a new root results. An index time stamp tree indexes this B-tree. It takes $O(\log(k))$ to search the index tree for the appropriate time stamp. In a B-tree all leaves are at the same level. Therefore a search operation takes, on an average, $O(\log(n)\log(m))$ to locate the node of interest. Let us suppose that a node in the B-tree has 80% node utilization, and the number of keys in a node justify a binary search. Then the search operation gives a processing order of $\log(n)(\log .80(m-1))$, where (m-1) is the maximum number of keys in a node of a B-tree of order “$m$,” and “$n$” is the number of nodes in the B-tree. Thus the total search time in this version management B-tree will be $lg(k)lg(n)(\log .80(m-1))$.

B-trees are by far the most frequently utilized data structures with which to implement databases. IBM’s VSAM is based on B-trees. So the path copying on a B-tree will provide version management for VSAM databases. Figure 7 depicts the proposed version management support for VSAM. The index tree, which points to different tree versions, can be maintained in primary memory. Once the tree version to be queried is located in the index tree, the appropriate page can be retrieved from the VSAM file on disk.
CONCLUSION

The paper provides an overview of version management databases. The version management systems are categorized as snapshot, rollback, historical, and temporal databases. The concept of persistence in balanced trees is utilized to create an efficient data structure with which to implement these temporal databases. The classical method of checkpointing proves extremely expensive in both memory usage and processing time. Path copying in a tree provides the needed persistence of historical data. A height-balanced(1) tree depicted the persistence characteristic. A persistent B-tree is proposed as an appropriate data structure to support version management on databases. A search operation on this persistent B-tree of order "m" is \( O(\log_2(m-1)) \), as all nodes along one path of an "m" way B-tree need to be replicated.

REFERENCES
