Versioning a Full-text Information Retrieval System

Peter G. Anick and Rex A. Flynn

Digital Equipment Corporation
111 Locke Drive, LMO2-1/D12
Marlboro, MA. 01752

ABSTRACT

In this paper, we present an approach to the incorporation of object versioning into a distributed full-text information retrieval system. We propose an implementation based on "partially versioned" index sets, arguing that its space overhead and query-time performance make it suitable for full-text IR, with its heavy dependence on inverted indexing. We develop algorithms for computing both historical queries and time range queries and show how these algorithms can be applied to a number of problems in distributed information management, such as data replication, caching, transactional consistency, and hybrid media repositories.

1 Introduction

Versioning has been an object of study in the fields of database and code management for many years, e.g., [DITTRICH88, STONEBRAKER87, KENT89, SNODGRASS90]. Perhaps because most research in Information Retrieval has been done on static data, this topic has received relatively little attention to date in the IR community. However, with the increasing interest in dynamic information environments, such as help-desk systems, in which textually represented knowledge is constantly being updated and augmented, the need for maintaining versioned data within IR systems may soon be growing considerably. In this paper, we show how the constraints of the full-text information retrieval task suggest a different solution from those employed so far for traditional structured database environments. We present an approach in which inverted indexes are augmented by delta change records such that "contemporary" queries suffer minimal performance degradation relative to an unversioned database and the performance of historical queries varies in proportion to the distance traveled back in time. Moreover, we show that the space impact of versioning indexes can be significantly reduced by using "partially versioned" index sets. We also explore how versioning can be exploited on behalf of a number of tasks required in a distributed, object-oriented IR system like AI-STARS [ANICK91]. Such tasks include the maintenance of transactional integrity, metadata evolution, data replication, cache management, and retrieval over hybrid media repositories.

This paper is organized as follows. We begin by motivating our interest in incorporating versioning into a dynamic help-desk environment and lay out our solution to the challenge of supporting temporal information within the space/time constraints typically imposed by full-text information retrieval applications. Next we explore how such versioning can be generalized to a distributed IR system like AI-STARS, which provides a layer of abstraction for querying and data replication above the level of physical repositories. We illustrate a number of applications that can be supported using our representation of temporal information - metadata evolution, data replication, transactional consistency, caching, and distributing temporal information across hybrid storage media. Finally, we compare our implementation with other approaches and discuss aspects of the problem which we have yet to explore.

15th Ann Int'l SIGIR '92/Denmark-6/92

2 Motivation and goals for versioning

In the corporate help-desk environment, problem-solving knowledge accumulates daily as new problems are solved and old remedies prove inadequate or obsolete. As a consequence, the on-line information base is constantly in flux; not only are new articles entered, but existing articles may be edited, annotated, or deleted. It can be useful in certain circumstances to know what information a customer may have received at some earlier date. Historical access to versioned data is one way to satisfy this requirement.

Also subject to change is the system's metadata. In applications like AI-STARS, which support the dynamic declaration of classes of information objects, class definitions may evolve over time in response to new administrative or retrieval needs. The need to support the graceful evolution of this metadata, allowing

instances of old classes to coexist (at least temporarily) with instances of the new classes, particularly in a distributed environment where changes propagate through the network at finite velocities, suggests a solution in which metadata is versioned as well.[1]

Such considerations, in addition to a number of other potential applications of temporal information in distributed information bases (see section 3), led us to investigate the incorporation of versioned objects into an information retrieval system. Our primary functional goals in extending our system with versioning were threefold:

• the ability to access historical versions of a given object,

• the ability to perform historical queries based on a "snapshot" of the database at some given point in time, and

• the ability to perform historical queries ranging over a specified time span.

There are other kinds of goals for versioning systems, as well as other kinds of goals for historical queries, which we are not trying to address:

- CAD/CAM and CASE tools have a very different notion of versioning [KATZ84, DITTRICH88]. These have to do, usually, with allowing multiple competing simultaneous views of the same data, and these views are usually explicitly generated by the users of the system. Tracking historical data, on the other hand, allows for "implicitly generated views," by specifying a particular time, but only *one* such view at any time.
- Other historical databases [STONEBRAKER87, ROWE87] promote the ability to query along the time dimension to equal status with the rest of the system capabilities. Although we want the capability for historical queries, our primary goal is to add this without impacting the performance of the typical "contemporary" query (which will continue to constitute the bulk of query activity in, for example, help-desk applications.)

Our performance goals are therefore as follows:

- There should be no (or minimal) additional cost for doing "contemporary" queries.
- The cost for doing historical queries (either "snapshot" or "time range") may be proportional to how far back in time the query goes.
- The space cost required should be proportional to the amount the data changes. If an object is added to the system, but is never modified, the space overhead for versioning should be minimal.

The last point is perhaps an important differentiator between a versioned information retrieval system and other kinds of versioned systems. Any individual article or metadata object, while subject to the possibility of extensive modification, is nevertheless in practice not likely to change very frequently or to a very great extent. Indeed, some of the collections in the AI-STARS application are essentially static, or disallow modifications to their members. We chose, therefore, to optimize performance for the most common cases - contemporary queries and "relatively" stable information objects.

We believe we have attained these goals in this design. An increase in code complexity as a result of versioning is inevitable. But we do not believe this translates to a significant increase in computational overhead.

In versioning a full-text information retrieval system, in addition to versioning the information objects, the index must be versioned as well. Since the notion of versioning objects is not new, most of our work has centered on designing an efficient versioned index scheme. For the purposes of this discussion, we will assume that the index contains only object set membership information, not concordance data. (However, we believe that versioned concordance data can also be handled within the proposed framework.)

2.1 Versioning the objects

Our algorithm for versioning the objects is essentially borrowed from [STONEBRAKER87]. Each version contains a timestamp identifying the time that the version was created. This timestamp serves (for us) as the version number. Only one version is stored in its entirety; the other versions store only the changes (i.e., difference or "delta" records) necessary to enable reconstructing the various time-based views of an object.

One minor difference with Stonebraker's model is that we propose allowing these "delta changes" (i.e., the versions which are only partially represented) as being able to proceed *back* in time (as well as forward in time) from the completely represented version. Since the latest view of the data is the one accessed most of the time, it normally makes sense to store the latest version completely and store delta changes backwards (but see section 3.7 below). This involves slightly more work at update time (existing blocks must be overwritten), but less work at retrieval time.

The other difference with Stonebraker's model is that we intend (under normal circumstances) to physically append the delta changes to the complete object representation. We rely on an underlying disk allocation scheme (segment-based), and disk I/O capabilities which will bring in a varying number of blocks in a single I/O. Appending the delta changes to the object therefore increases the transfer time from disk, but otherwise incurs no other overhead, so long as the delta changes are not accessed in memory.

Thus, the computational cost required to reconstruct an historical view of an object is proportional to how far back in time one wants to go (how many delta change records are applied to the latest view). Our informal experience with this approach so far is that the cost of "rolling back" an object to its historical view is not significant. This method also satisfies the space overhead goal; the additional overhead from versioning, when an individual object only has a single version, is a timestamp stored with

[1] [AHLSEN84] offers a similar rationale for versioning metadata in a distributed object-oriented database.

it. This timestamp is arguably useful anyway (as it encodes the creation date). With a good difference algorithm, as long as the changes to objects are small, the size of delta change records ought to be small.

The algorithm for constructing an historical view of an object is as follows:

- Start with the most current view of the object.
- For each delta change record following the object, which has a timestamp greater than the timestamp for the requested view (in descending time order), apply the differences in the delta change record against the "current view" of the object, changing the object to the way it looked before that time.

This algorithm is a form of "rollback." Historical databases are sometimes called "rollback databases" [SNODGRASS90], because, effectively, every single database transaction that ever occurs may be rolled back.

2.2 Versioning the indexes

Unlike most conventional database systems, full-text information retrieval systems usually construct fully inverted indexes of the contents of textual fields to assure adequate retrieval performance. By assigning object ids in an ascending sequence of integers, sets of objects can be efficiently represented and manipulated in memory as bitmaps and also effectively compressed on disk using a variety of data compression techniques (e.g., [BOOKSTEIN90, SALTON89]). It was our goal to preserve these valuable properties, while augmenting indexes with versioning information to support historical queries.

One alternative is to include with each object id in an index its time span(s) of applicability. However, this "natural" representation is not amenable to bitmap operations, nor is it very space efficient, considering the number of object ids likely to occur in each index set. A second alternative, more in line with our approach to versioning the objects, is to store each index entry as a bitmap set, indicating the objects that satisfy the index at the current time, augmented by a set of delta change records that trace the incremental changes to the index backwards over time.

A delta change record for an index entry would look like the following:

[timestamp, object id, {+/-}],

where the timestamp is the time the change occurred to the index entry, a "+" indicates that the object id was added to the index entry at that time, and a "-" indicates that the object id was removed from the index entry.

By the same argument put forth for the storing and retrieval of versioned objects, the I/O overhead for storing index data this way ought not to be significantly different, and the computational overhead in constructing an historical view of the index entry in memory is proportional to how far back in time one wants to go.

However, there are two problems with this representation. The first is due to the fact that index entries change far more frequently than individual objects do in an IR system. It is therefore the case that the amount of data involved in tracking historical views of the index entries can be huge. In fact, with this naive implementation, there is at least one delta change record for every object that is a member of the set in the index entry, i.e., the "+" record indicating when the object first becomes a member of the index set. This naive implementation is therefore prohibitively costly in terms of space - at least one timestamp would have to be stored for every object/index entry combination.

The second problem is that, although such an index representation works in a fairly obvious way for performing a query for a specific time in the past (one constructs an historical view of each index entry for each term in the query), the implementation of a query over specific object versions in a time *range* is not so obvious.

In the following sections, we will present a variation on this approach which alleviates the space problem, and show how it can be used to perform historical queries. We will then show an algorithm for performing time range queries which operates directly on this representation.

2.3 Partially versioned sets

To meet our performance goal of minimal space overhead for the case where every object in the database has only a **single** version, the representation of an index set under this boundary condition should contain just the bitmap set of ids and no delta change records.

This can be accomplished if an index entry is updated according to the following rules:

- If it is the "first version" of an object that is being indexed, only update the bitmap, and do not add a ("+") delta change record.
- Otherwise, if it is a later version, update the bitmap, and append a delta change record indicating how the index entry changed as a result of the object modification.

Such a representation scheme satisfies our space requirements well. While the initial addition of an object to a database is likely to affect perhaps thousands of index entries, any subsequent object modification is likely to affect but a few. Therefore, although delta change records in the index are not nearly as efficiently represented (given that they have timestamps stored with them), the number of them ought to be small relative to the size of the index.

We will call a set constructed according to the above rules a "partially versioned" set, as it lacks certain categories of temporal information. If such a set is "rolled back" to an historical view (by applying the delta change records in reverse), the fact that the delta change records for the *first* versions of objects are missing means that the sets will be left with objects as members even at times prior to when the objects had come into existence. Fortunately, the information about the times at which objects come into existence is common across all the index entries, and can therefore be stored in a single, separate list. This list can be

represented as timestamp/object id pairs, can be sorted in descending timestamp order, and can be traversed at the same time the delta changes for an index entry are traversed, to eliminate from consideration any objects erroneously left as members of the set.

2.4 Versioned database sets

As noted above, the use of partially versioned index sets requires maintaining a single shared list of information about the objects in a database. Such a list is independently useful. Dynamic non-versioned IR systems often maintain a global set of "current" database members to distinguish between object ids of current objects and those of objects that have been deleted. Global sets can also be used to subdivide a physical database into logical subsets within a single id space (see section 3.1.2).

We will use the term "versioned database set" (or simply "database set") to refer to the analogue of this global set as employed in our versioned database mechnanism. A versioned database set contains a list of *specific object versions* that are members of a database. Like index sets, we represent such a database set as a contemporary bitmap plus a sequence of delta change records. The critical difference between the two representations is that the database set contains "complete" information about versioning and can therefore be used on its own, whereas the index entry sets do not contain complete information and must be interpreted in conjunction with their corresponding database set, as described above.

The delta records in a database set (which we will refer to as a "fully versioned" set) are of the following form:

[timestamp, object id, op]

where the timestamp indicates the time at which the change occurred to the set, the object id indicates the object affected, and op is an operation on the set. There are three operations:

- "+", which indicates that the object was added to the set at that time.
- "-", which indicates that the object left the set at that time.
- "=", which indicates that the object was a member of the set before the time, and is still a member of the set after the time, but a new version of the object occurred at the time. It is not immediately obvious why the "=" record is required; we will explain its value later when we consider the algorithm for performing a time range query.

2.5 Algorithms for versioned set update and access

In this section, we show how the versioned set representations described above for the database sets and index sets can be (1) updated and (2) accessed with respect to a given point in time.

2.5.1 Updating the database set

Once the decision to update a database has been made, the algorithm for updating the database set is as follows. There is one case corresponding to each of the delta change record operation types. Note that for databases for which object membership is based on satisfying some content-based filter, any one object may go "in" and "out" of the database multiple times, as successive versions of the object satisfy or fail to satisfy the filter that defines the database.

- If an object version is to become a member of the database and the object is not currently a member, add its id to the set of object id's for the database, and add a delta change record to the front of the set of delta change records, using the timestamp for the object version, and the "+" operation indicator.
- If an object version is to become a member and the prior object version is currently a member, make no change to the set of object id's, and add an "=" delta change record.
- If an object leaves the database, remove the object from the set of object id's, and add a "-" delta change record indicating the time the object leaves.

2.5.2 Accessing the database set historically

A view of the database set as of any particular time may be reconstructed as follows:

- Make a "working copy" of the set of object id's representing the latest view of the database set.
- (Working through the delta change records in decreasing temporal order) for each delta change record whose timestamp is greater than the desired time of access, update the working copy as follows:
  - if it is a "+" record, remove the object id from the working copy.
  - if it is a "-" record, add the object id to the working copy.
  - if it is an "=" record, ignore it.

2.5.3 Updating an index entry set

Updating index entries can be performed by an indexer process reacting to changes in the object versions. The indexer process must deal with a number of different situations:

- If an object is becoming a new member of the database, all index entries which the object's content satisfies must be updated to include the object as a member.
- If an object is already a member of the database, the indexer need only change those index entries that are affected by the change. (Recall that "=" records never need to be stored in the partially versioned index entries.) The indexer thus needs to calculate what is the "same" and what has changed. This can be done either by examining the delta change records in the object, or by running an indexing pass over the old version of the object, in memory, and over

the new version of the object, in memory, and only updating indexes based on the differences.

- If an object is leaving membership in the database, then all index entries that refer to the object as of its most recent prior version must be updated. These can be identified by running an indexing pass over the most recent prior version of the object, in memory.

With respect to the updating of any one individual index entry, there are three possible circumstances:

- If the object is being added to the index entry and to the database at the same time, add the object to the set of object id's for the index entry (but do not create a delta change record).
- If the object is being removed from the index entry (whether or not it is also being removed from the database at the same time), remove the object from the set of object id's for the index entry, and add a "-" delta change record for the change.
- If the object is being added to the index entry but was already a member of the database set, add the object to the set of object id's, and add a "+" delta change record for the change.

As mentioned earlier, two kinds of delta change records are deliberately omitted from the index entry sets, as compared with the database sets. No "=" records are stored; and no "+" records are stored when an object is being added to the database simultaneously to being added to the index entry. Eliminating the need to store such records for all index entries results in considerable space savings.

2.5.4 Accessing an index set

Given a time at which the view of the index set needs to be reconstructed:

- Construct the view of the database set as of that time (in the manner described above).
- Make a copy of the latest set of object id's in the index entry.
- For each delta change record in the index which has a timestamp greater than the desired time of access:
  - If the delta change record indicates a "+", remove the object from the copied set of object id's.
  - If the delta change record indicates a "-", add the object to the copied set of object id's.
- Restrict the resulting set against the database set constructed in the first step. Since no object will be a member of the database set before it was created, this serves to remove erroneously remaining objects from the result set.

2.5.5 Performing an historical query

Given a particular access time at which a query is to be executed, historical index entry sets can be reconstructed for that time (by rolling back the delta changes, as described above) for each term in the query. Then the standard bitmap Boolean operations can be performed on these reconstructed sets. Note that the historical database set need only be reconstructed once, and can be re-used for filtering each index entry.

2.5.6 Performing time range queries

In the scheme above for performing historical queries, it is not necessary to figure out during the query processing exactly *which* object version it is that satisfies the query. In a "snapshot" or "rollback" database, it is sufficient to know that the query can be accurately calculated for the particular time. The retrieval of the appropriate object versions, needed, for example, in displaying a title list, can be carried out subsequently by reconstructing the objects as of the same historical access time. Thus, the query processor need only return an object id, not an object version.

With a query spanning a time *range*, however, information must be known about individual object versions, because multiple object versions may exist (and may satisfy the query) within the range. The output of such a query must therefore be, in some sense, a set of object *versions*, not just a set of objects[1]. As it turns out, the versioned database set representation *is* a (complete) representation for a set of object versions. One could, for example, transform the delta change representation into an object-centered representation, in which each object version is paired with the set of time spans for which it is a member of the database.[2] However, we can use the database set and index set representations in approximately their current form to calculate a range-based query. Intuitively, one can think of the algorithm we will propose as a way of re-calculating the query (but efficiently) at every point in time that some event occurred in the database, within the time range specified.

The result set which this range query algorithm will create has two components: (1) a bitmap set of object id's, which indicates the set of object versions which satisfy the query at the end of the time range, and (2) a set of delta change records, which indicate how this set of object versions changes as one goes back through time to the start of the time range. We have already solved the problem of creating the bitmap set, for this task is identical to that of executing an historical query on the input at the end of the time range. The algorithm we are about to describe therefore focuses on how to compute the second component of the result set, the set of delta change records. In a manner analogous to how a historical query is performed, the task of generating the set of delta change records breaks down into two subtasks. The first subtask is reconstructing the full set of *changes* that occurred to each index

[1] In our definition of a time range query, a specific object version satisfies the query either completely or not at all. There are other more complicated kinds of time-based queries which we are not addressing in this design. (For example, one might make a query where one object version might satisfy one clause in the query, and a different object version might satisfy another clause in the query.)

[2] This transformation algorithm is, however, beyond the scope of this paper.

entry set in the input between the end and the start of the time range, and the second subtask is performing the equivalent of a Boolean query using the reconstructed index entry *changes* as the leaves of the query tree. It may not be obvious that Boolean operations can be performed directly on a pair of *changes* sets to generate another *changes* set. The next section explains how our representation makes this possible. In the section following, we show how the first subtask, that of filling out the set of delta changes in each index entry, is accomplished.

2.5.6.1 Query processing over delta change sets

In processing a query using delta change information, we will assume that the input is in the form of a binary Boolean query tree with a set of delta change records at each leaf. The goal of the algorithm is to construct another set of delta change records at the root, this set encoding the set of object versions that satisfy the query at some point within the time range specified. This task reduces via recursion to the subtasks of performing (time-extended) Boolean AND and OR operations on a pair of sets of delta change records, and inverting a set of delta changes for a Boolean NOT operation.

It is at this point that the need for the "=" delta records in fully versioned sets is manifested, for it is the case that the set of delta change records for a leaf node (even with the "+" records which were deliberately dropped from the index sets reinserted) is insufficient for performing the binary Boolean operations. As an example, assume that no changes have occurred to one index entry in the time range, but in another index entry, a particular object left membership of the index entry set within the time range, and that the Boolean operation being performed on the two sets is an OR:

index entry 1 (delta changes in time range only): (none)

index entry 2 (delta changes in time range only): [timestamp t, object id x, -]

It is impossible to determine whether the result of the Boolean operation should include this object beyond the time of its change or not, without examining membership in the first index entry. If the object *was* a member of the first index entry throughout the range, the "event" of the object leaving membership of the second would be immaterial to the result. On the other hand, if it was not a member of the first index entry in this time range, the object would leave the result set at the same time that it left index entry 2.

It is in order to deal with this ambiguity that "=" delta change records have been introduced. We must guarantee that if the object had been a member of index entry 1, there would have been a corresponding record in the delta changes for index entry 1, as follows:

index entry 1 (delta changes in time range only): [timestamp t, object id x, =]

In other words, the object's version has changed, but it remained a member of the database. On the other hand, if there is no corresponding record in the delta changes for the timestamp t/object id x combination, that indicates that the object was not a member of the index entry at that time.

Since we order the delta changes in the same way (by timestamp, by object id) for every set, binary Boolean operations may proceed stepwise through any two sets of delta changes, as follows:

- Start with the first delta change record in each set.
- Attempt to perform a comparison between two delta change records at a time, as follows:
  - If two delta change records are being compared and they do not correspond (i.e., neither timestamp nor object id matches),
    * pick the later/higher object id of the two
    * assume, from the absence of a complementary delta change record in the other set, that the object was not a member of that set before the change, and is still not a member of the set after the change. We represent this case in our diagrams as a "<>" record, even though such a record does not physically exist in the set representation.
    * perform the appropriate Boolean comparison against the virtual "<>" record (see figure 1 below).
    * move to the next location in the set with the later/higher id delta change record, and perform the next comparison.
  - If the two delta change records do correspond,
    * perform the appropriate Boolean comparison against both records,
    * move to the next location in both sets, and perform the next comparison.
  - If one of the lists is exhausted
    * assume a "<>" delta change record for the corresponding timestamp/object id combination, as before.
    * perform the appropriate Boolean comparison against the imaginary "<>" record, as before.
    * move to the next location in the set with more values, and perform the next comparison.

For each of the binary operations, there are sixteen possible combinations of circumstances in the comparison, and four possible outcomes, if we consider one of the outcomes (the "<>" case) as being the decision not to generate a corresponding delta change record for the result set. It is easiest to consider the Boolean operations as tables (shown in figure 1 below). The computation can be conceived as separately performing the Boolean operation on membership both before and after the "change." Every delta change operation corresponds to exactly one combination of before/after membership, shown in parentheses to make the computation clearer.

It is not possible to interpret a unary NOT outside of a context. It makes no sense, for example, when an object has been added to an index entry, and the object has been added to a database at the same time, to convert the "+" record into a "-" record under a NOT operation. The inverse in this case is that the object did not become a member of the index entry, i.e., "<>". We therefore re-cast every unary NOT as a binary AND NOT operation against the set of database delta change records for the same time range (i.e., database set AND NOT negated set). We therefore show the table for the Boolean AND NOT operation as well. Notice that a number of the circumstances in this table can never actually show up (these have been marked by *'s), because the index set delta change records must have been created in the context of the database set.

2.5.6.2 Computing fully versioned delta change sets for index entries

Recall that, while the previous algorithm requires fully versioned sets, the index sets initially at the leaf nodes of the query tree are only partially versioned sets. Fully versioned sets can be generated for each index entry by using a combination of the delta changes for the database set in the time range, the delta changes for the index entry set in the time range, and a copy of the set of object members in the index entry at the end of the time range. (The latter set must be constructed anyway, for the historical query at the end of the time range.)

Every delta change record in the index entry set within the time range of the query has a corresponding delta change record in the database set, since initial membership and every version change is recorded in the versioned database set. The converse is, however, not true. So the task of this algorithm is to "fill in" some of the occasions where a corresponding index delta change record is missing.

The possible combinations of circumstances are enumerated in figure 2. We will continue to use the virtual "<>" operator for clarity, although such records need not be physically included in the set.

There are two situations where "<>" delta change information causes ambiguity. The first is when an object becomes a member of the database. It may or may not become a member of the index set at the same time. The second is when the object version changes, but there is no corresponding effect on the index entry (it does not leave or enter membership of the index entry set). In this case, it may "remain" either a member, or not a member of the index entry. Finally, there is one situation where the "actually missing" information causes no ambiguity: the object leaves membership of the database. If it does not leave membership of the index set at the same time, then it must not have been a member of the index set (and it still is not, since the index entry is defined on the database).

AND	+ (01)	- (10)	= (11)	<>(00)
+ (01)	+ (01)	<>(00)	+ (01)	<>(00)
- (10)	<>(00)	- (10)	- (10)	<>(00)
= (11)	+ (01)	- (10)	= (11)	<>(00)
<>(00)	<>(00)	<>(00)	<>(00)	<>(00)*

OR	+ (01)	- (10)	= (11)	<>(00)
+ (01)	+ (01)	= (11)	= (11)	+ (01)
- (10)	= (11)	- (10)	= (11)	- (10)
= (11)	= (11)	= (11)	= (11)	= (11)
<>(00)	+ (01)	- (10)	= (11)	<>(00)*

AND NOT	+ (01)	- (10)	= (11)	<>(00)
+ (01)	<>(00)	+ (01)*	<>(00)*	+ (01)
- (10)	- (10)*	<>(00)	<>(00)*	- (10)
= (11)	- (10)	+ (01)	<>(00)	= (11)
<>(00)	<>(00)*	<>(00)*	<>(00)*	<>(00)*

Each delta change record type is followed by parentheses indicating membership in the set respectively before and after the delta change record was encountered. For example, "+" is followed by (01), meaning that the corresponding object was not a member of the set before this delta change record, but is afterwards. The "<>" is a placeholder indicating that no matching record exists, and the object was not a member of the set before or after this time.

The AND NOT table is used to process unary NOT operations against the set of delta changes in the database. It reads as if the database set were represented in the row heads, and the negated set is represented in the column heads. Impossible combinations are marked with an "*".

Figure 1. Boolean operations on "matching" delta change records.

Database set change type	Index entry set change type	Entry missing?	Explanation
+	<>	yes	The object became a member of the database at this time, but not a member of the index set
+	+	yes	The object became a member of the database at this time, and a member of the index set
=	<>	yes	The object was already a member of the database, but a new version of it still is not a member of the index set
=	+	no	The object was already a member of the database, and a new version of it caused it to be added to the index set.
=	=	yes	The object was already a member of the database, but a new version of it is still a member of the index set.
=	–	no	The object was already a member of the database, and a new version of it caused it to be removed from the index set.
–	<>	yes	The object left the database, but was not a member of the index set.
–	–	no	The object left the database, and left the index set.

Figure 2. Completing the index entry delta changes.

This table shows the possible combinations of circumstances that can occur between any two "matching" delta change records, one in the index entry and one in the database set. The "<>" change record type again is used as a placeholder. The second column describes what the index entry would contain, if the delta changes had been completed. The third column indicates whether this delta change record would actually be missing in a partially versioned set of delta changes.

Both ambiguous situations can be resolved by looking at the set of members of the index entry calculated for the time of the database set delta change record. However, rather than recomputing this set each time the ambiguity is encountered, the set of members in the index entry can be calculcated efficiently at the same time the delta change records are traversed. Here, then, is the algorithm, starting with the set of delta change records for the database set in the time range, the set of delta change records for the index entry set in the time range, and the set of members of the index entry at the end of the time range. The assumption is that both delta change sets are in descending timestamp/object id order:

For each successive delta change record in the database set, look at the next delta change record in the index set.

- If they correspond (have the same timestamp/object id), update the set of index entry members as follows:
  - if the index delta change record is a "-" record, add the object id to the set.
  - if the index delta change record is a "+" record, remove the object id from the set.
- If they do not correspond (one is missing from the index entry delta changes set):
  - if the database set delta change record is a "-" record, do nothing (conceptually, add in a "<>" record).
  - if the database set delta change record is an "=" record,
    - if the object is a member of the index entry set, insert a corresponding "=" record into the index list.

* if the object is not a member of the index entry set, do nothing.

if the database set delta change record is a "+" record,

* if the object is a member of the index entry set, insert a corresponding "+" record into the index list.

* if the object is not a member of the index entry set, insert a corresponding "=" record into the index list.

At the same time, remove the object id from the member set.

Once this is done, the updated delta change record list for the index entry is "fully versioned".

2.5.6.3 Review of the range query algorithm

This section summarizes the steps involved in the range query algorithm. It assumes a previously constructed query tree of Boolean nodes, a time range of validity, a database for the context of the query, and index entries for the database at each leaf of the query tree.

1. Perform an historical query using the tree of Boolean nodes, as of the *end* of the time range for the query. Save the output as the bitmap portion of the result set. Also save copies of each index bitmap set that was rolled back in the computation of the historical query.

2. Extract the subset of delta change records that occur in the database set between the end and start times of the query time range.

3. Extract the subsets of delta change records that occur in each index entry set between the end and start time of the query time range.

4. Turn each index entry's subset of delta change records into a fully versioned set by stepping through the previously extracted change records from the database set, and updating the copy of the index entry set saved in the first step (as described in section 2.5.6.2).

5 Compute a new fully versioned set of delta change records for each successively higher node in the query tree, starting at the leaves (as described in section 2.5.6.1).

6. Append the delta changes to the bitmap result set computed in the first step.

3 Applications of temporal versioning in distributed Information Retrieval

In this section, we introduce the architecture for distributed information retrieval that we are implementing in AI-STARS, and show how the incorporation of temporal versioning may be applied to a number of problem areas.

3.1 AI-STARS

The AI-STARS project is an on-going research program at Digital Equipment Corporation, investigating methods for improving enterprise-wide full-text information retrieval for Digital's Customer Support organization. The practical demand for very rapid access to data combined with limitations in network speed and reliability dictate that highly used data be replicated locally. Thus, our target environment must mix distributed access to data with partial data replication. To respond to these information needs, the AI-STARS architecture incorporates (1) a self-describing meta-model, in which user-defined classes of information are themselves represented as information objects to better accommodate heterogeneous databases, schema evolution and data distribution, and (2) a level of database abstraction for querying and replicating data above the level of physical repositories.

3.1.1 The AI-STARS data model

Specialists utilize a wide variety of types of information in their problem solving, from bulletin board notices and symptom-solution articles chronicling previous experience to crash dump summaries and software problem reports. Rather than force all articles in the database into a single format (thereby losing the ability to query on specialized fields), AI-STARS supports a heterogenous database in which new classes of information objects can be defined on the fly. As in SMALLTALK [GOLDBERG83], the data model is self-describing; fields and classes are themselves full-fledged data objects. In this way, database administrators can query and browse the metadata just like any other data. Furthermore, in a distributed system in which data is replicated on multiple sites, the distribution of meta-information can be carried out in exactly the same manner as for any other information object.

3.1.2 Collections

One of the problems that inevitably arise once an information retrieval system begins to make a large volume of distributed information available is that, without a useful partitioning of the information space, the user may be at a loss as to which databases to open to satisfy an information need. In the WAIS [KAHLE91] and Project Mercury (CMU) distributed retrieval systems, users can choose from a menu of available sources, where these sources correspond to physical repositories of data, each typically representative of some area of interest. In many corporate environments, however, organizing the retrieval space along the lines of physical repositories is inappropriate, since the way

that data is partitioned for the purposes of administration may be quite different from the way that data should be organized for retrieval. Moreover, even if the physical databases do conform to some useful conceptual classification, it is rare that any one classification scheme can account for all the ways that users would like to carve up the information space. In order to overcome this limitation, the AI-STARS architecture incorporates the construct of a "collection" as a virtual aggregation of information objects defined by either (1) explicit inclusion or (2) the (recursive) application of a query filter to the union of other collections.

We will use the term "repository collection" to refer to a physical database, comprising all the objects created and maintained at a specific site. Taken together, all the repository collections in a distributed application form a partition of the set of objects available to that application. We will use the term "computed collection" to refer to a set of objects assembled by the application of a query to a domain composed of one or more collections. There can be any number of computed collections and the same object may be a member of more than one of them.

Computed collections must do more than just calculate their contents. They must also present "virtual" index entries, so that a query can be performed over them as if they were "real databases." This can be accomplished in a straightforward manner once the "collection set" has been computed. One can recursively look up the index entries in the collections from which the computed collection is derived, union these together, and use the collection set to eliminate any objects from the result which are not also members of the computed collection.

It is also possible to physically store computed collections, replicating the virtual database, so as to avoid the need to "go back to the source" of the data at query time. Such replication would normally be done when the data is distant, and the cost of computing the collection at query time would be prohibitive. The obvious way of doing this is storing what is computed. This includes

- the objects
- the index entries
- the "set of objects" that are part of the collection.

As this data is identical to what needs to be stored for a repository collection, retrieval and update on a replicated collection need be no different from retrieval and update on a repository collection.

3.2 Meta-data evolution

As noted above, in the AI-STARS data model, classes and fields are themselves objects. As such, a database administrator can augment the schema for an application interactively, without the need for recompiling and/or relinking software. Typically, such metadata is maintained in one or more metadata collections; the new field and class objects are simply distributed as data wherever these collections are replicated across the WAN. As [AHLSEN84] points out, versioning of metadata is useful in distributed object oriented databases to allow for the graceful evolution of the schema, since it permits instances of an old class version to be updated to the new class definition over time rather than all at once. In fact, instances of the old version need never be updated, as both the new and old class definitions remain available to interpret instances.

In AI-STARS, instances are linked to their class definitions via an instance_of field, whose value contains both the object id of its class and the timestamp of the class version. This ensures that when the object is accessed, it can be interpreted (e.g., displayed) with respect to the proper class version. It also allows the query engine to retrieve all instances of a class, regardless of the version of the class, or, alternatively, only those objects that are instances of a specific class version. By using the set difference operation, a database administrator can identify the set of instances of the class that have not yet been updated to the new class version, a useful way to check on the status of an evolving database, in which updating instances may take time and effort.

3.3 Computed collections

A computed collection abstraction can be built on top of the previously described historical and time range algorithms:

- The set of object members in the collection can be calculated, either for a particular time, or over a time range, by executing the query that defines the collection appropriately.
- Any index entry for a collection may be calculated by unioning the corresponding index entries for the domain collections together, and restricting the result against the set of objects in the computed collection. As with the set of object members in the collection, the index entry may be calculated at a particular time (by doing recursive historical lookups), or it may be calculated for a time range, by doing recursive time range lookups, and using the extended time range Boolean operations to perform the appropriate unions and restrictions.
- Finally, a view of any object version in the collection can be presented by looking through the domain collections for one collection that has the object version as its member, and retrieving the object version recursively from that collection.

3.4 Replicated collections

In AI-STARS, we must replicate collections to provide the kind of response time and reliability that help-desk clients expect. Replication, however, comes at a cost; in addition to the storage cost involved in replicating data, the replicated data may be out of date.[1] Our replication scheme is based on a schedule, i.e., periodically a background process is initiated which will check

[1]To see why this is so for our architecture, one must consider what the alternative would be; whenever an object is added or modified in a repository, it would have to be tested for membership against the queries in all of the replicated collections that might include it, to see if the corresponding replications should be updated.

against the sources for a particular collection and copy any relevant changes over. It turns out that updating data on approximately a nightly basis is acceptable to most AI-STARS users, although the replication scheme as we specify it has greater flexibility.

There are two ways that collection replication can proceed. A particular repository where a collection is replicated (we call this replicated collection an "incarnation" of that collection) may go to another collection incarnation that is more up-to-date, and find the changes that have occurred; this is no more than the set of delta changes that have been stored in the collection set for the source incarnation, between "now" and the last time the check occurred. Alternatively the collection incarnation may compute the changes by going to sources for the domain collections over which its query is recursively defined. Our time range query algorithm provides the solution to this; the query is applied over the time between now and the last checked time.

Since we have subscribed to being able to perform distributed queries in this system, both of the methods described may operate without requiring additional data distribution channels, if the schedule is kept with the repository associated with the replicated collection, and a process fired up by that repository performs the network retrieval. In other words, a "pull" model of data distribution fits nicely with our distributed architecture.

One problem with replicating collections on a schedule is that, depending on where the replication process goes to perform its query, the degree that the data is out-of-date depends not only on the periodicity with which the local process is fired up, but also the periodicity with which the data in its "source" is updated. As a consequence of this, we have the database administrator specify, not how frequently the data in a replicated collection incarnation is checked, but how out-of-date the data in that incarnation is allowed to get.[1]

Subscribing to a degree of "out-of-dateness" is far more useful to an AI-STARS user than a frequency of checking. For example, we are planning on allowing the user to specify as part of a query how "out-of-date" the query is allowed to get. With this information, the system can go out to the database and try to find the closest incarnation of a collection which satisfies the user's requirements.

3.5 Transactional consistency

One of the concerns of most database management systems is to preserve a consistent view of the database for the time span of a transaction. The purpose, in a system where multiple users may update the database simultaneously, is to ensure that the user work with a view of the database that is unaffected by the changes of other users. In a temporal database like AI-STARS, one can achieve transactional consistency by choosing a time at which to start the transaction and then viewing the database as of this time for the duration of the transaction. The natural place to start a transaction is the start of a query. If this is done, the title list, the objects that are displayed, any query reformulation and any interobject traversal all use the same view of the data as the initial query.

Transactional consistency is greatly complicated by having to deal with different collection incarnations which are out-of-date to differing degrees. Since collections may be recursively defined on other collections, it is quite possible (and invisible to the querier) that a particular computed collection ends up being based on two different collection incarnations that overlap in their contents, but with different temporal views (because of differing out-of-dateness) of the overlapping data.

Versioning our collections provides us with opportunities for solving this problem that would be unavailable otherwise. The simplest solution is to find the latest time that is shared among the source data at the start of the transaction. If the querier has specified an "out-of-dateness" requirement for the query, as described in the previous section, (and the system has found incarnations which satisfy the requirement,) the latest shared time will satisfy the querier's requirements. The transaction is then defined as accessing the database at this time, and the data will be viewed at this time for the duration of the query.[2]

3.6 Collection caching

One of the motivations for developing the concept of a computed collection[3] is that the results of computing a collection are identical for multiple queriers, and may therefore be usefully cached. These caches may reside on collection "compute servers" which compute the results of one or more collections, have lots of memory and horsepower, and are available on the network as a shared resource.

From a client perspective, accessing a compute server for a collection is no different than accessing a replicated collection via a server. In fact, collection replication is a form of disk caching, and some of the same algorithms can be used to implement both. The principal difference is that, in a cached collection, some of the data is not kept locally (and the subset in the cache may vary.) The cache equivalents to the collection replication algorithm are as follows:

- Somehow it is determined that the cache is out of date.

[1] Algorithms to enforce this are necessarily approximate. However, the failure modes are a matter of degree, can be detected, and can be incorporated into any decision-making.

[2] We have also explored alternatives where we relax the consistency requirement, so that the querier can see the "latest view available" of parts of the data, even if other parts are not valid until the same time. Our versioning scheme can handle this, by allowing the transaction to specify multiple access times, each valid for a subset of the data considered in the transaction.

[3] as an alternative to our original notion of a "stored query" [ANICK91].

- A time range query is made against the domain collections to find the changes that have occurred since the last time the "database" was checked.
- The returned results (which will be a list of delta changes) are used to invalidate object versions in the cache that have changed.

In an historical database, the format of cache invalidation turns out to be somewhat different from that in traditional databases. An object version is never truly "invalid;" rather, it may become an historical version. This applies to the collection sets and the index entry sets as well. The goal with the cache for an historical database is therefore to mark the information that has been updated as historical, and let it disappear from the cache in the same way that any other unreferenced data disappears. In this way, historical data is treated identically to any other data in the cache (and may stay in the cache if it is accessed frequently).

As with collection replication, we plan on the client detecting the "out-of-dateness." This can be checked at the start of every read transaction (see the previous section). Alternatively, if the information in the cache is within the user's "out-of-dateness" requirements, we can avoid performing a check until the next transaction is initiated where the requirements are not met.

3.7 Hybrid media collections

Within Digital, it is becoming commonplace to collect together information on a CD-ROM periodically, make many copies of it, and distribute it. We have been considering such a scenario as an alternative for distributing data in AI-STARS, and as a mechanism for making AI-STARS data available to queriers who are not completely connected to our internal electronic network.

The periodicity of distributing CD-ROM's is likely to be on the order of once every one or two months. As a consequence, for those collections which are dynamic, we have been working with the notion of placing any updates to the CD-ROM collections on a local disk. Access to the collection would be resolved by considering the CD-ROM and the disk together. Since the volume of updates should be small (and would be integrated back into the CD-ROM periodically), such a scheme might be viable for smaller computer systems.

Our versioning methods provide a natural way of storing and applying incremental changes; delta change records can be stored on disk for objects and indices, and can be applied to the collection on the CD-ROM to "fix it up", as it is accessed. The major difference with what we have described so far is that the delta changes are changes *forward*. Since we are trying to save space on the disk, we would store only the information necessary to bring a CD-ROM index entry or object up-to-date (i.e., we would not duplicate information already on the CD-ROM).

The representation and the algorithm for applying delta changes *forward* for an object need be no different, and the solution in this case is identical to Stonebraker's. Unfortunately, it turns out that the partial delta change list representation is asymmetrical, and cannot be used as it stands for recording forward delta changes. This problem is resolved if we never omit storing a "+" delta change record for an index entry. Doing so incurs a higher overhead for the index entry, but only temporarily so, until the next CD-ROM distribution comes along.[1]

4 Discussion

4.1 Related work

The principal inspiration for the research presented here has been the work by Stonebraker et al. on POSTGRES [STONEBRAKER87], a relational database with object-oriented extensions. Stonebraker and his colleagues have spent some time looking at algorithms that are appropriate for versioning indexes in a temporal database.

The simplest such algorithm appends [Tmin - Tmax] time ranges to each entry in the index, to record the time validity of the entry. For us, this would mean that each object id in an index entry would have a [Tmin - Tmax] range associated with it. Such a representation is functionally equivalent to our fully versioned set representation, and all of the Boolean operations could be performed on it, by unioning and intersecting the [Tmin - Tmax] time ranges.

A more sophisticated variation on this is the R-Tree algorithm [GUTTMAN84, KOLOVSON89, KOLOVSON90]. R-Trees are an extension to B-Trees, which cluster data on disk according to multiple dimensions simultaneously. Time can be used as one of these dimensions. Whereas B-Trees perform node merge and split operations with respect to data distributed over one dimension, R-Trees perform the corresponding merge and split operations according to how data distributes over multiple dimensions. The advantage of clustering data along multiple dimensions is, for example, that in performing a time range query, the system need neither consider (traverse) all of the events that occurred in a particular time range, nor consider (traverse) all of the time-based events that occurred to a particular range of objects. The [Tmin - Tmax] representation, by contrast, clusters along the object dimension first, i.e., it forces all time information to be considered for a range of objects. Our delta change record representation, on the other hand, clusters along the time dimension first, i.e., it forces events to be considered for all objects in a particular time range, whether or not all of these events are relevant.

[1]There are other more complicated representational alternatives which preserve the space characteristics for the index entries better. Essentially, the problem is that the set of objects in the index entry cannot be updated (on the CD-ROM) directly, and it contains information that is used in the "rollback" algorithm. So the idea for the solution is to store the required updates on the set as a separate list. This optimization may, however, not be worth it.

Which of these index representations is most attractive depends on the nature of the application. For instance, a system in which any one object's values change a great deal on average, and where the accessor is equally likely to access historical data as current data, would not be served well by the indexing scheme proposed here. In such a system, the space benefit of omitting index delta change records for the first versions of objects would disappear, as would the benefit of keeping our delta records in descending historical time order.

We believe, however, that for a textual information retrieval system our versioning representation is much superior. Since indexes take up far more of the total space in an IR system, it is very expensive to have time ranges associated with every entry in an index. Furthermore, we believe that, although objects will change in our database, they will not change very much, and the vast majority of the index entries will be created with the first version of every object.

4.2 Computational complexity

So long as we use a bitmap representation for the member sets in memory, the computational complexity of the historical query algorithm depends on the following components:

- the number of events in the database set between the "current time" and the time of the query. This is the maximal number of delta change records that need to be considered for the database set, and for each of the index entry sets.
- the number of leaf nodes in the query tree. This will be the number of index entry sets that must be considered.
- the number of internal nodes in the query tree. This will determine the number of Boolean operations performed.
- the size of the bitmaps. In principle, this is the number of objects in the database.

The number of internal nodes and number of leaf nodes in the query can be considered to be approximately the same. Therefore, the computational cost is:

O(([# historical events considered] + [# objects in the database]) * [# nodes in query])

As described previously, as the [# historical events considered] goes to 0, the computational cost becomes the same as that for a non-versioned Boolean query.

It turns out that the computational complexity of a range query is no different. To see why this is so, we must break down the range query into its two components, the historical query at the end of the time range, and the process of completing the delta changes in the index entries, and then performing the Boolean operations on the delta changes. The factors here are the number of historical events in the time range (used to complete the delta changes, and to determine the size of each delta change list), and again, the number of nodes in the query (which represents both the number of steps in the query, and the number of index entries considered). The cost of a range query is therefore:

O(([# historical events to end of range] + [# objects in the database]) * [# nodes in query]) +

O([# historical events in time range] * [# nodes in query])

If these two factors are added together, this is identical to the cost of an historical query at the beginning of the time range.

One significant difference in computational complexity between our algorithms and, for example, the [Tmin - Tmax] versioned set representation is that using bitmaps for Boolean operations makes the size of the database a consideration, rather than the size of an individual index entry. IR systems typically make this tradeoff because performing Boolean operations on bitmap sets is very fast. Our design has been oriented this way, and so long as the number of historical events considered is not excessive, our historical/time range queries ought to proceed very quickly also.

4.3 Further work

Each of the applications described in section 3 requires more study and refinement. For example, we do not yet know to what degree collection compute servers could obviate the need for physically replicated collections. There are many possible ways of extending the notion of "out-of-dateness" of replicated collections/queries to more powerful schemes by which an AI-STARS user could make the response-time/up-to-dateness trade-off. And algorithms for enforcing "up-to-dateness" of particular replicated collections will inevitably evolve.

Since the information bases currently in use by Digital's Customer Support Specialists are not yet versioned, we have had no direct experience with the practical consequences of versioning over long periods of time. It may or may not be necessary, for example, to develop disk "vacuuming" schemes [STONEBRAKER87].

While we have not addressed the issue of versioning concordance information here, this is clearly an issue that merits further study, as many of the operations expected of a full-text information retrieval system depend on concordance data (see, e.g., [BURKOWSKI92]). Note that even a small change to an object (such as the insertion of a single word into the text) can have a significant ripple effect on the concordance.[1]

Finally, while we have presented how underlying support for historical and snapshot querying may be implemented, we have not yet investigated how best to put these capabilities into the hands of the end-user. The range of issues to be addressed here include: how to incorporate time operators in a query language [GADIA88], how to present a notion of time in the user interface, and how to present a title list for a range query.

[1] This can be mitigated to some extent by padding the numerical assignment of concordance locations at structural boundaries.

5 Conclusions

The trend toward the use of full-text information retrieval in dynamic information environments such as help-desk systems has motivated us to explore strategies for object versioning that are sensitive to the time and space constraints of the information retrieval task. We have presented a design for a versioned IR system that adds considerable architectural functionality and flexibility while minimizing the impact on retrieval performance and storage space. The algorithms for historical and time range query can be applied to a number of problems in the design of an object-oriented distributed system in which "logical" views of distributed data may be computed and stored.

ACKNOWLEDGEMENTS

The evolution of the ideas embodied in AI-STARS and its implementation have been a group effort. Jeffrey Robbins was the principal designer and implementor of the original STARS system. Jeff, along with Bryan Alvey, Norman Lastovica, and James Wagner, of Digital's Colorado Springs Customer Support Center, are collaborating with Intelligent Information Applications Development Group members Suzanne Artemieff, Jong Kim, Jim Moore, Clark Wright and the authors on the development of AI-STARS. The entire team's contributions have been essential for the creation and realization of the ideas presented in this paper.

REFERENCES

[AHLSEN84] Ahlsen, M. , A. Bjornerstedt, A. Britts, S. Hulten, and L. Soderlund. An Architecture for Object Management in OIS. ACM Transactions on Office Information Systems, 2(3), 1984.

[ANICK90] Anick, P. G., J. D. Brennan, R. A. Flynn, D. R. Hanssen, B. Alvey and J. M. Robbins. A Direct Manipulation Interface for Boolean Information Retrieval via Natural Language Query, in Proceedings of ACM/SIGIR '90, Brussels, 1990.

[ANICK91] Anick, P. G., R. A. Flynn, and D. R. Hanssen. Addressing the Requirements of a Dynamic Corporate Textual Information Base, in Proceedings of ACM/SIGIR '91, Chicago, 1991.

[BOOKSTEIN90] Bookstein, A. and S. T. Klein. Construction of Optimal Graphs for Bit-Vector Compression, in Proceedings of ACM/SIGIR '90, Brussels, 1990.

[BURKOWSKI92] Burkowski, F. J. An Algebra for Hierarchically Organized Text-Dominated Databases, to appear in Information Processing and Management.

[COOMBS90] Coombs, J. H. Hypertext, Full-Text, and Automatic Linking, in Proceedings of ACM/SIGIR '90, Brussels, 1990.

[CROFT87] Croft, W. B. and R. T. Thompson. I^3R: A New Approach to the Design of Document Retrieval Systems. Journal of the American Society for Information Science, 38, 1987, pp. 389-404.

[DITTRICH88] Dittrich, K. R. and R. A. Lorie. Version Support for Database Systems, IEEE Transactions on Software Engineering, Vol. 14, no. 4, April 1988.

[GADIA88] Gadia, Shashi K. A Homogeneous Relational Model and Query Languages for Temporal Databases, ACM Transactions on Database Systems, Vol. 13, No. 4. Dec. 1988, pp. 418-448.

[GOLDBERG83] Goldberg, A. and Robson, D. Smalltalk-80: The Language and Its Implementation. Addison-Wesley, 1983.

[GUTTMAN84] Guttman, A. R-Trees: A Dynamic Index Structure for Spatial Searching, in Proceedings of ACM/SIGMOD, Boston, 1984.

[KAHLE91] Kahle, B. and A. Medlar. An Information System for Corporate Users: Wide Area Information Servers, WAIS Corporate Paper version 3, April 1991.

[KATZ84] Katz, Randy H. and Lehman, Tobin J. Database Support for Versions and Alternatives of Large Design Files, IEEE Transactions on Software Engineering, Vol. SE-10, no. 2, March 1984, pp. 191-200.

[KENT89] Kent, William, An Overview of the Versioning Problem. in Proceedings of 1989 ACM SIGMOD Conference on the Management of Data, 1989, pp. 5-7.

[KOLOVSON89] Kolovson, Curtis and Stonbraker, Michael. Indexing Techniques for Historical Databases. Memorandum No. UCB/ERL M89/34, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, Apr. 1989.

[KOLOVSON90] Kolovson, Curtis and Stonbraker, Michael. S-Trees: Database Indexing Techniques for Multi-dimensional Interval Data. Memorandum No. UCB/ERL M90/35, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, Apr. 1990.

[KHOSHAFIAN86] Khoshafian, S. N. and Copeland, G. P. Object Identity. ACM Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, Sept. 1986.

[ROWE87] Rowe, I., and Stonebraker, M. The Postgres Data Model. Proceedings of the XIII International Conference on Very Large Databases, Brighton, England, September 1987. Morgan Kaufman Publishers, San Mateo, CA.

[SALTON89] Salton, G. Automatic Text Processing: the Transformation, Analysis, and Retrieval of Information by Computer. Addison-Wesley, 1989.

[SNODGRASS90] Snodgrass, Richard (ed.) Temporal Databases, Status and Research Directions, ACM SIGMOD Record, Vol. 19, no. 4, Dec. 1990, pp. 83-97.

[STONEBRAKER87] Stonebraker, M. The Design of the POSTGRES Storage System. Proceedings of the XIII International Conference on Very Large Databases, September 1987. Morgan Kaufmann Publishers. San Mateo, CA.