Performance Evaluation of a Temporal Database Management System

Ilsoo Ahn and *Richard Snodgrass*†

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27514

Abstract

A prototype of a temporal database management system was built by extending Ingres It supports the temporal query language TQuel, a superset of Quel, handling four types of databases static, rollback, historical and temporal A benchmark set of queries was run to study the performance of the prototype on the four types of databases We analyze the results of the benchmark, and identify major factors that have the greatest impact on the performance of the system We also discuss several mechanisms to address the performance bottlenecks we encountered

1. Introduction

Database management systems are supposed to model reality, but conventional DBMS's lack the capability to record and process time-varying aspects of the real world With growing sophistication of DBMS applications, the lack of temporal support raises serious problems For example, conventional DBMS's cannot support *historical queries* about the past status, much less *trend analysis* which is essential for applications such as decision support systems [Ariav 1984] There is no way to represent *retroactive* or *postactive* changes, while support for *error correction* or *audit trail* necessitates costly maintenance of backups, checkpoints, or transaction logs to preserve past states There is also a growing interest in applying database methods for *version management* and *design control* in computer aided design, requiring capabilities to store and process time dependent data Without temporal support from the system, many applications have been forced to manage temporal information in an ad-hoc manner

This research was supported by NSF grant DCR-8402339
† The work of this author was also supported by an IBM Faculty Development Award

The need for providing temporal support in DBMS's has been recognized for at least a decade [Bubenko 1976, Schueler 1977] Recently, the rapid decrease of storage cost, coupled with the emergence of promising new mass storage technologies such as optical disks [Fujitani 1984, Hoagland 1985], have amplified interest in database management systems with temporal support or version management [Copeland 1982, Wiederhold 1984] A bibliographical survey contained about 70 articles relating time and information processing [Bolour et al 1982], at least 30 more articles have since appeared in the literature However, most efforts on temporal databases have focussed on conceptual aspects such as modeling, query languages or semantics of time Little has been written on implementation issues, let alone performance analysis of such systems, except for a few version management systems [Katz & Lehman 1984, Svobodova 1981], rollback DBMS's [Ariav & Morgan 1982, Copeland & Maier 1984, Lum et al 1984], and an earlier version of LEGOL 2 0 [Jones et al 1979]

In this paper, we discuss the implementation and performance of a prototype temporal DBMS, and identify major factors that have the greatest impact on the performance of the system Sections 2 and 3 briefly set the context for this investigation, describing the types of databases in terms of temporal support, and the query language supported by the prototype The next two sections

outline the implementation and provide a comprehensive analysis of the performance of the prototype Finally, Section 6 discusses several mechanisms that address the performance bottlenecks identified in the prototype

2. Types of Databases

Numerous schemes have been proposed to record and process history data augmented with additional time attributes. A taxonomy of time to characterize the time attribute and define types of database management systems in terms of temporal support was recently proposed [Snodgrass & Ahn 1985, Snodgrass & Ahn 1986]

	No Rollback	Rollback
Static Queries	Static	Rollback
Historical Queries	Historical	Temporal

Figure 1: Types of Databases

As summarized in Figure 1, two orthogonal criteria are the capability to make *historical queries* about the past status of the enterprise modeled by a database, and the ability to *rollback* to the past state of the database modeling an enterprise The former concerns the progression of events through time, the latter concerns the recording of those events in a database *Historical* databases support historical queries, incorporating *valid time* *Rollback* databases support rollback operations, incorporating *transaction time* *Temporal* databases support both operations, incorporating both kinds of time A third kind of time is *user-defined* time, whose semantics is defined by each application program Supporting user-defined time requires only minimal changes to a DBMS, and does not substantially impact its performance

Historical databases record the history of the enterprise being modeled, and view tuples *valid* at some moment as of *now* Rollback databases record the history of database activities, and view stored tuples, whether valid or not, *as of* some moment in time Temporal databases combine the benefits of the two, viewing tuples *valid* at some moment seen *as of* some other moment Further examples that emphasize the often subtle differences in these four types of databases are described elsewhere [Snodgrass & Ahn 1986]

3. TQuel

TQuel (Temporal QUEry Language) [Snodgrass 1986], a superset of Quel [Held et al 1975], supports both historical queries and rollback operations It extends several Quel statements to provide query, data definition and data manipulation capabilities supporting all four types of databases It expresses historical queries by augmenting the **retrieve** statement with the **when** predicate to specify temporal relationships among participating tuples, and the **valid** clause to specify how the implicit time attributes are computed for result tuples The rollback operation is specified by the **as of** clause for the rollback or the temporal DBMS's The **append**, **delete**, and **replace** statements were augmented with the **valid** and the **when** clauses in a similar manner Finally, the **create** statement was extended to specify the type of a relation, whether static, rollback, historical or temporal, and to distinguish between an interval and an event relation if the relation is historical or temporal The semantics of TQuel was formalized using tuple relational calculus [Snodgrass 1986]

The example query in Figure 2 inquires the state of a database **as of** 1981, shifting back in time Retrieved tuples satisfy not only the **where** clause, but also the **when** clause specifying that the two tuples must have coexisted at some moment The **valid** clause specifies the values of the *valid from* and *valid to* attributes of the result tuples

```
retrieve (h.id, h.seq, i.id, i seq)
   valid from start of (h overlap i)
         to end of (h extend i)
   where h id = 500  and
         i amount = 73700
   when  h overlap i
   as of "1981"
```

Figure 2: A TQuel Query

4. A Prototype Temporal DBMS

It involves substantial research and implementation effort to fully integrate temporal support into the DBMS itself New access methods and query processing algorithms need to be developed to achieve reasonable performance for a variety of temporal queries, without penalizing more frequent non-temporal queries

As an intermediate step towards a fully integrated temporal DBMS, we built a prototype by modifying portions of the static DBMS Ingres [Stonebraker et al 1976], while still keeping the conventional access methods and query processing algorithms Hence the performance of the prototype was expected to be less than ideal, rapidly deteriorating for both temporal and non-temporal queries However, it is still useful to identify problems with conventional access methods and query processing algorithms, and to suggest possible mechanisms for addressing those problems In addition, the prototype can serve as a comparison point for fully integrated DBMS's developed in the future

The prototype supports all the augmented TQuel statements **retrieve, append, delete, replace** and **create** The **valid, when** and **as of** clauses are fully supported, though default values for these clauses are not supplied The **copy** statement was modified to perform batch input and output of relations having time attributes It also supports all four types of databases static, rollback, historical and temporal

The parser was modified to accept TQuel statements and generate an extended syntax tree with subtrees for **valid, when,** and **as-of** clauses Some of the query evaluation modules were changed to handle the newly defined node types and implicit time attributes such as *valid from, valid to, transaction start* and *transaction stop* Functions to handle temporal operators **start of, end of, precede, overlap**, and **extend** were added in the one-variable query processing interpreter

The system relation was modified to support various combinations of implicit time attributes, which depend on the type of a relation as specified by its **create** statement A time attribute is represented as a 32 bit integer with a resolution of one second It has a distinct type, so that input and output can be done in human readable form by automatically converting to and from the internal representation Various formats of date and time are accepted for input, and resolutions ranging from a second to a year are selectable for output

One of the most important decisions was how to embed a four-dimensional temporal relation into a two-dimensional static relation as supported by Ingres There are at least five such embeddings [Snodgrass 1986] Our prototype adopts the scheme of augmenting each tuple with two *transaction time* attributes for a rollback and a temporal relation, and one or two *valid time* attributes for a historical and a temporal relation depending on whether the relation models events or intervals

For a rollback relation, an **append** operation inserts a tuple with the *transaction start* and *transaction stop* attributes set to the current time and "**forever**" respectively A **delete** operation on a tuple simply changes the transaction stop attribute to the current time A **replace** operation first executes a **delete** operation, then inserts a new version with the transaction start attribute set to the current time A historical relation follows similar steps for **append, delete** and **replace** operations with the *valid from* and *valid to* attributes as the counterparts of transaction start and transaction stop attributes Values of the valid from and valid to attributes are defaulted to the current time and "**forever**" respectively, but also can be specified by the **valid** clause

For a temporal relation, an **append** operation inserts a tuple with the transaction start of the current time, and transaction stop of "**forever**" Attributes valid from and valid to are set as specified by the **valid** clause, or defaulted if is is absent A **delete** operation on a tuple sets the transaction stop attribute to the current time indicating that the tuple was virtually deleted from the relation Next a new version with the updated valid to attribute is inserted indicating that the version has been valid until that time A **replace** operation first executes a **delete** operation as above, then appends a new version marked with appropriate time attributes Therefore, each **replace** operation in a temporal relation inserts two new versions This scheme has a high overhead in terms of space, but completely captures the history of retroactive and postactive changes In addition, all modification operations for rollback and temporal relations in this scheme are *append only,* so write-once optical disks can be utilized A more detailed discussions of these operations can be found elsewhere [Snodgrass 1986]

The prototype was constructed in about 3 person-months over a period of a year; this figure does not include familiarization with the Ingres internals or with TQuel Most changes were additions, increasing the source by 2,900 lines, or 4 9 % (our version of Ingres is approximately 58,800 lines long)

5. Benchmarking the Prototype

We define the *update count* for a tuple as the number of update operations on the tuple, and the *average update count* for a relation as the average of the update counts over all tuples in the relation We hypothesized that, as the average update count increases, the performance of our prototype with conventional access methods would deteriorate rapidly not only for temporal but also for static queries We postulated the major factors to affect the performance of a temporal DBMS were the type of a database, the query type, the access methods and loading factors, and the update count

A benchmark was run to confirm these hypotheses in various situations, and to determine the rate of performance degradation as the average update count increased. This section describes the details of the benchmark, presents its results, and analyzes the performance data from the benchmark

5.1. The Benchmark

To compare performance on different types of databases, we needed test databases of all four types described in Section 2 For each of the four types, we created two databases, one with a 100% loading factor and the other with a 50 % loading factor As the sample commands for a temporal database in Figure 3 show, each database contains two relations, *Type_h and Type_i, where Type is one of* `Static, Rollback, Historical, and Temporal`

```
create persistent interval Temporal_h
        (id = i4, amount = i4,
        seq = i4, string = c96)
modify   Temporal_h to hash on id
        where fillfactor = 100

create persistent interval Temporal_i
        (id = i4, amount = i4,
        seq = i4, string = c96)
modify   Temporal_i to isam on id
        where fillfactor = 100
```

Figure 3: Creating a Temporal Database

Type_h is stored in a hashed file, and *Type_i* is stored in an ISAM file The loading factor of the files are specified with the **fillfactor** parameter in a **modify** statement [Woodfill et al 1981]

Each tuple has 108 bytes of data in four attributes, **id**, **amount**, **seq** and **string** Id, a four byte integer, is the key in both relations Amount and **string** are randomly generated as integers and strings respectively, and **seq** is initialized as zero In addition, rollback and historical relations carry two time attributes, while temporal relations contain four time attributes Attributes transaction start and valid from are randomly initialized to values between Jan 1 and Feb 15 in 1980, while attributes transaction stop and valid to are set to "**forever**" indicating that they are the current versions The evolution of these relations will be described shortly

Each relation is initialized to have 1024 tuples using a **copy** statement The page size in our prototype is 1024 bytes With 100 % loading, there are 9 tuples per page in static relations, and 8 tuples per page in rollback, historical, or temporal relations Therefore, we need at least 114 pages for each static relation, and 128 pages for each of the others The actual size depends on the database type, the access method, the loading factor, and the average update count

Twelve sample queries with varying characteristics comprise the benchmark as shown in Figure 4 These queries were chosen in an attempt to represent the characteristic queries in databases with temporal support, to isolate the effects of various TQuel clauses, to exercise the access methods available in Ingres, and to demonstrate the possibility of performance enhancement The number of output tuples were kept constant regardless of the update count, except for queries Q01, Q02 and Q12

Q01 retrieves all versions of a tuple *(version scan)* from a hashed file given a key Q03 is a *rollback* query, applicable only to rollback and temporal databases, retrieving the state of a relation as of some moment in the past Q05 retrieves the most recent version of a tuple from a hashed file given a key, while Q07 retrieves the most recent version of a tuple from a hashed file without a key, resulting in a sequential scan of the whole file

Queries Q02, Q04, Q06 and Q08 are counterparts of Q01, Q03, Q05, and Q07 respectively, where even numbered queries access an ISAM file and odd ones access a hashed file Both Q09 and Q10 join current versions of two relations, Q09 goes through the primary access path of a hashed file and Q10 goes through an ISAM file

```
    range of h is temporal_h
    range of i is temporal_i

Q01: retrieve (h.id, h.seq)
    where  h.id = 500
Q02: retrieve (i.id, i.seq)
    where  i id = 500

Q03: retrieve (h.id, h.seq)
    as of  "08.00 1/1/80"
Q04: retrieve (i.id, i seq)
    as of  "08:00 1/1/80"

Q05: retrieve (h.id, h seq)
    where  h.id = 500
    when   h overlap "now"
Q06· retrieve (i.id, i.seq)
    where  i.id = 500
    when   i overlap "now"

Q07  retrieve (h.id, h.seq)
    where  h.amount = 69400
    when   h overlap "now"
Q08. retrieve (i.id, i seq)
    where  i.amount = 73700
    when   i overlap "now"

Q09: retrieve (h.id, i.id, i.amount)
    where  h.id = i.amount
    when   h overlap i and
           i overlap "now"
Q10· retrieve (i id, h id, h.amount)
    where  i id = h.amount
    when   h overlap i and
           h overlap "now"

Q11. retrieve (h.id, h.seq, i.id,
           i.seq, i amount)
    valid from start of h
           to end of i
    when   start of h precede i
    as of  "4:00 1/1/80"

Q12. retrieve (h.id, h.seq, i id,
           i.seq, i.amount)
    valid from start of (h overlap i)
           to end of (h extend i)
    where  h.id = 500  and
           i.amount = 73700
    when   h overlap i
    as of  "now"
```

Figure 4: Benchmark Queries

Queries Q05 through Q10 all refer to only the most recent versions They are termed *static* queries in the sense that they retrieve the current state of a database as if from a static database. For a static database, the **when** clause in these queries are neither necessary nor applicable For a rollback database, we use the **as of** clause instead of the **when** clause For example, **when x overlap "now"** will become **as of "now"**

Q11 is a query involving a *temporal join,* a join of two tuples based on temporal information In this query, the **as of** clause specifies the *rollback* operation shifting the reference point to a past moment The **when** clause specifies a temporal relationship between two versions, where the value of valid from attribute in the version from *Type_*h relation is earlier than the corresponding value in the version from *Type_*i relation The **valid** clause specifies that transaction start attribute of the result tuple be set to the value of transaction start attribute in the version from *Type_*h relation, and that transaction stop attribute of the result tuple be set to the corresponding value in the version from *Type_*i relation Q12 contains all types of clauses in TQuel, inquiring the state of a database as of "now" given both temporal and non-temporal constraints Obviously, Q11 and Q12 are relevant only for a temporal database

These twelve queries were run on each of eight test databases as described earlier, two databases, with the loading factor of 100 % and 50% respectively, for each of `Static`, `Rollback`, `Historical`, and `Temporal` We focused solely on the number of disk accesses per query at a granularity of a page, as this metric is highly correlated with both CPU time and response time There are a few pitfalls to be avoided with this metric Disk accesses to system relations are relatively independent of the database type or the characteristics of queries, but more dependent on how a particular DBMS manages system relations Also, the number of disk accesses varies greatly depending on the number of internal buffers and the algorithm for buffer management To eliminate such variables, which are outside the scope of this paper, we counted only disk accesses to user relations, and allocated only 1 buffer for each user relation so that a page resides in main memory only until another page from the same relation is brought in

Once performance statistics were collected for all sample queries, we simulated the uniformly distributed evolution of a database by

incrementing the value of **seq** attribute in each of the current versions. The time attributes were appropriately changed for this **replace** operation using the default of **valid from "now" to "forever"** as described in Section 4. Thus a new version (two new versions for temporal relations) of each tuple is inserted, and the average *update count* of the database is increased by one Performance on the sample queries were measured after determining the size of each relation appended with new versions This process was repeated until the average update count reached 15, which we believed high enough to show the relationship between the growth of I/O cost and the average update count The benchmark was run on a Vax 11/780, consuming approximately 20 hours of CPU time

5.2. Performance Data

Space requirements in various databases were measured as the average update count ranged from 0 to 15, and were useful for analyzing the I/O cost measured in the benchmark Figure 5 shows the data for the average update count of 0 and 14 along with the *growth per update* It also shows the *growth rate,* obtained when dividing the growth per update by the size for the update count of 0 From this table, we find that

- The rollback and the historical databases have the same space requirements
- The temporal database consumes the same amount of space as the rollback and the historical databases for the update count of 0
- The temporal database, following the embedding scheme described in Section 4, requires almost twice the additional pages as the update count increases
- The growth per update for a hashed file varies slightly due to key collisions in hashing

I/O costs for sample queries on each database were measured as the average update count increased from 0 to 15 Output costs account for storing temporary relations, which remain constant because the size of temporary relations is the same for the sample queries regardless of the update count Since they are negligible compared with the input costs, being 56 pages for Q09 and Q10 each and 4 pages for Q12 on the historical and the temporal databases, and 0 for the others, we concentrate on the analysis of the input costs Figure 6 shows the input costs for the temporal database with 100% loading

Similar tables, a total of 8, were obtained for each database of different types and loading factors We summarize the input costs for sample queries on various databases with the average update count of 0 and 14 in Figure 7

Figure 7 shows that the rollback and the historical databases exhibit similar performance, while the temporal database is about twice more expensive than rollback and historical databases for the update count of 14 If we draw a graph for the input costs shown in Figure 7, we get Figure 8 (a) Figure 8 (b) is a similar graph for the rollback database with 50% loading, showing jagged lines caused by the odd numbered updates filling the space left over by the previous updates before adding overflow pages

5.3. Analysis of Performance Data

The graphs in Figure 8 show that input cost increases almost linearly with the update count, but with varying slopes for different queries A question is whether there are any particular relationships independent of query types between the input cost and the average update count, and between the input cost and the database type To answer this question, we now analyze how each sample query is processed, and identify the dominant operations which can characterize each query

Though queries Q01 and Q05 are functionally different from each other, one being the *version scan* and the other a *static query,* our prototype built with conventional access methods uses the same mechanism to process them Both queries are evaluated by accessing a hashed file given a key *(hashed access)* Likewise, Q02 and Q06 requires the access to an ISAM file given a key *(ISAM access)* Queries Q03, Q04, Q07 and Q08 all need to scan a file, whether hashed or ISAM *(sequential scan)*

Processing Q09 first scans an ISAM file sequentially doing selection and projection into a temporary relation *(one variable detachment)* It then performs one hashed access for each of 1024 tuples in the temporary relation *(tuple substitution).* Here the dominant operation is the hashed access, repeated 1024 times Q10 is similar to Q09 except that the roles of the hashed file and the ISAM file are reversed Hence the dominant operation for Q10 is the ISAM access

Q11 is evaluated by sequentially scanning one file to find versions satisfying the **as of** clause

For such a version, the other file is sequentially scanned for versions satisfying both the **as of** clause and the **when** clause Here the dominant operation is the sequential scan Processing Q12 requires a sequential scan and a hash access to find versions satisfying the **where** clause, then joins them on time attributes according to the **when** clause Since the number of versions extracted for the join is small enough to fit into one page each, the dominant operation is the sequential scan

From this analysis, we can divide the input cost into the *fixed* portion and the *variable* portion The fixed cost is the portion which stays the same regardless of the update count It accounts for traversing the directory in the ISAM, or for creating and accessing a temporary relation whose size is independent of the update count The variable cost is defined to be the result of subtracting the fixed cost from the cost of a query on a database with no update Operations contributing to the variable cost will grow more expensive as the number of updates on the relation increases

Now we can define the *growth rate* of the input cost on a database with the update count of n as .

$$Growth\ Rate_n = \frac{C_n - C_0}{(variable\ cost) \times n}$$

where

C_n = *input cost for update count of n*
C_0 = *input cost for update count of 0*

The growth rate is the key aspect of an implementation, characterizing the performance degradation as the update count increases Clearly the ideal would be a growth rate close to 0

Fixed costs, variable costs and growth rates for sample queries on various types of databases were calculated The growth rate was relatively independent of the update count n, as indicated by the linearity shown in Figure 8 Figure 9 shows fixed costs, variable costs and growth rates for sample queries on the rollback and the temporal databases with the loading factor of 100% and 50% each The historical database shows the same variable costs and the growth rates as the rollback database, except for Q03 and Q04 which are not applicable to historical databases But its fixed costs are the same as the temporal database, except for Q03, Q04, Q11 & Q12 which are not applicable

Rather surprisingly, the growth rate turned out to be independent of the query type and the access method as far as access methods of sequential scan, hashing or ISAM are concerned. It was, however, dependent on the database type and the loading factor For example, the growth rates for operations such as sequential scan, hashed access, and access of data pages in ISAM are all 2 0 in case of the temporal database with 100% loading On the other hand, the growth rates for similar operations are approximately 0 5 in case of the rollback or the historical database with 50% loading

From these analyses, we can make several observations as far as access methods of sequential scan, hashing or ISAM are concerned

- The fixed and the variable costs are dependent on the query type, the access method and the loading factor, but relatively independent of the database type
- The growth rate is approximately equal to the loading factor of relations for rollback or historical databases
- The growth rate of input cost is approximately twice the loading factor of relations for temporal databases
- The growth rate is independent of the query type and the access method

The fact that the growth rate can be determined given the database type and the loading factor without regard to the query type or the access method has a useful consequence From the definition of the growth rate, we can derive the following formula for the cost of a query when the update count is n

$$\begin{aligned} C_n &= C_0 + (growth\ rate) \times (variable\ cost) \times (n) \\ &= (fixed\ cost) + (variable\ cost) + \\ &\quad (growth\ rate) \times (variable\ cost) \times (n) \\ &= (fixed\ cost) + (variable\ cost) \times \\ &\quad [\,1 + (growth\ rate) \times n\,] \end{aligned}$$

Therefore, when the cost of a query on a database with the update count of 0 is known and its fixed portion is identified, it is possible to predict future performance of the query on the database when the update count grows to n Note that the fixed cost, and hence the variable cost, can even be counted automatically by the system, except when the size of a temporary relation varies greatly depending on the update count

5.4. Non-uniform Distribution

Thus far, we have assumed uniform distribution of updates where each tuple will be updated an equal number of times as the average update count increases Since the assumption of uniform distribution may appear rather unrealistic, we also ran an experiment with a non-uniform distribution To simulate a maximum variance case, only 1 tuple was updated repeatedly to reach a certain average update count We measured performance of queries on the updated tuple and on any of remaining tuples, then averaged the results weighted by the number of such tuples Since it takes $O(n^2)$ page accesses to update a single tuple for n times, owing to the overflow chain ever lengthening, we repeated the process only up to the update count of 4, which was good enough to confirm our subsequent analysis

Performance of a query is highly dependent upon whether the tuple participating in the query has an overflow chain We hypothesized that updating tuples with a high variance would affect the growth rate significantly, owing to the presence of long overflow chains for some tuples and the absence of such chains for others However, the growth rate averaged over all tuples turned out to remain the same as the uniform distribution case For example, if we update one tuple in a temporal relation 1024 times, the average update count becomes one For a query like Q01, a hashed access to any tuple sharing the same page as the changed tuple costs 257 page accesses, while a hashed access to any tuple residing on a page without an overflow costs just one page access Therefore, the average cost becomes three page accesses, the same as the uniform distribution case

We can extend this result to a more general case If the number of primary pages is x with 100% loading, there will be approximately $2x$ overflow pages for the average update count of one in a temporal relation Let y be the number of primary pages which have overflow pages, and z be the number of primary pages which do not have an overflow, then $y + z = x$ Since the average length of overflow chains is $\frac{2x}{y}$ pages, the average cost of a hashed access to such a relation will be

$$\frac{y \times (\frac{2x}{y} + 1) + z \times 1}{y+z} = \frac{y}{x} \times \frac{2x}{y} + \frac{y+z}{x} = 3$$

showing the same result as the more restricted case discussed above

This reasoning can be generalized for other database types, access methods, loading factors, query types, and update counts in a similar fashion Now one more observation about the growth rate can be added

- The growth rate is independent of the distribution of updated tuples

We conclude that the results from the benchmark we ran under the assumption of uniform distribution are still valid for any other distribution

6. Performance Enhancement

As the results of the benchmark indicate, sequential scans are expensive Access methods such as hashing and ISAM also suffer from rapid performance degradation due to ever-growing overflow chains Reorganization does not help to shorten overflow chains, because all versions of a tuple share the same key

Since lower loading reduces the number of overflow pages in hashing and ISAM, it results in a lower growth rate Hence better performance is achieved with a lower loading factor when the update count is high But there is an overhead for maintaining a lower loading factor, which may cause worse performance than a higher loading when the update count is low Lower loading requires more space for primary pages Scanning such a file sequentially (*e g* for query Q07 or Q08) is more costly Especially for ISAM, lower loading requires more directory pages, which may increase the height of the directory For example, query Q10 for the update count of 0 in Figure 7 reads in 3385 pages with 50% loading, significantly higher than 2233 pages with 100% loading

We conclude that access methods such as hashing or ISAM are not suitable for a database with temporal support There are other access methods that adapt more gracefully to dynamic growth, such as B-trees, dynamic hashing, extendible hashing, and grid files [Nievergelt & Hinterberger 1984] These methods require complex algorithms and significant overheads to maintain certain structures as new records are added But the performance is still dependent on the count of all versions, which may be significantly higher than the count of current versions Furthermore, a large number of versions for some tuples will require more than a bucket for a single key, causing similar problems exhibited in conventional hashing and ISAM It is also difficult to maintain secondary indices for these methods, which often

split a bucket and rearrange its records, and to utilize write-once storage medium like optical disks Therefore, new storage structures and access methods tailored to the particular characteristics of temporal databases are needed to enhance performance significantly

Databases with temporal support maintain both the current and the history data on line But the current and the history data exhibit clear differences in their characteristics, such as the number of versions, storage requirements, access frequency and update patterns These differences make it natural to process them separately exploiting their unique characteristics Therefore, we adopt the *two level store* with separate storage areas for the current and the history data [Ahn 1986] The *primary store* contains current versions which can satisfy all non-temporal queries and possibly some of frequently accessed history versions The *history store* holds the remaining history versions This scheme to separate current data from the bulk of history data can minimize the overhead for non-temporal queries, and at the same time provide a fast access path for temporal queries

In addition, queries retrieving records through non-key attributes (*e g* Q07 and Q08) can be facilitated by *secondary indexing* There are several alternative structures for a secondary index on a relation with multiple versions The index may be stored into a single file for all the versions *(1 level),* or may itself be maintained as a *2-level* structure having a current index for the current data and a history index for the history data In each case, any storage structure such as the heap, hashing or ISAM may be chosen for the index

Figure 10 shows the estimated input costs for the sample queries on the temporal database with the two level store and the secondary indexing, where '—' indicates no change from the conventional case The advantage of the two level store is evident in processing static queries such as Q05 through Q10 The cost remains constant for any update count As shown under the column *Simple* in Figure 10, Q10 on the temporal database with the update count of 14 costs 2233 pages instead of 34493 pages Version scan (Q01 and Q02) can also be improved by clustering history versions of the same tuple into a minimum number of pages, *e g* 28 history versions into 4 pages as the column *Clustered* in Figure 10 shows

Columns under *as 1-Level* in Figure 10 show the estimated input cost when an index is maintained as a single file on the amount attribute for the temporal relation The index needs eight bytes for each entry, four for the secondary key and four for a tuple id, and hence can store 101 entries in a page of 1024 bytes Since there are 29 versions multiplied by 1024 tuples, 295 pages are needed for the index If we store them in a heap, it costs 324 pages, 295 index plus 29 data pages, to evaluate Q07 This is in fact more expensive than the simple 2-level store without any index, though better than the conventional structure itself If we use hashing for the index, the cost is reduced to 30 page accesses with 1 index page and 29 data pages

If we use the 2-level indexing with a separate index for current data, there are only 1024 entries in the index, requiring 11 index pages Q07 costs 12 pages with the heap index, while it costs only 2 pages with the hashed index, as shown in columns under *as 2-Level* in Figure 10 Note the difference between 3717 pages and 2 pages for processing the same query

7. Summary

We built a prototype of a temporal database management system by extending the static DBMS Ingres It supports the temporal query language TQuel, a superset of Quel, handling all four types of databases static, rollback, historical and temporal A benchmark with a set of queries was run to study the performance of the prototype on the four types of databases with two loading factors We analyzed the results of the benchmark, and identified major factors that have the greatest impact on the performance of the system As far as the access methods of sequential scan, hashing or ISAM are concerned, the growth rate is determined by the database type and the loading factor, but independent of the query type, the access method, or even the distribution of updated tuples A formula was obtained to estimate the cost of a query on a database with multiple temporal versions, when the cost of a query on the database with a single version is known and its fixed portion is identified We also discussed possible performance enhancements using two-level storage structures and secondary indexing mechanisms tailored for databases with temporal support

8. Bibliography

[Ahn 1986] Ahn, I *Towards an Implementation of Database Management Systems with*

Temporal Support In *The Second International Conference on Data Engineering,* IEEE Feb 1986, pp 374-381

[Ariav & Morgan 1982] Ariav, G and H L Morgan *MDM Embedding the Time Dimension in Information Systems* Technical Report 82-03-01 Department of Decision Sciences, The Wharton School, University of Pennsylvania 1982

[Ariav 1984] Ariav, G *Preserving The Time Dimension In Information Systems* PhD Diss The Wharton School, University of Pennsylvania, Apr. 1984

[Bolour et al 1982] Bolour, A , T L Anderson, L J Debeyser and H K T Wong *The Role of Time in Information Processing A Survey SigArt Newsletter,* 80, Apr. 1982, pp 28-48

[Bubenko 1976] Bubenko, J A , Jr *The temporal dimension in information modeling* Technical Report RC 6187 #26479 IBM Thomal J. Watson Research Center Nov 1976

[Copeland 1982] Copeland, G *What If Mass Storage Were Free? IEEE Computer,* 15, No 7, July 1982, pp 27-35

[Copeland & Maier 1984] Copeland, G and D Maier *Making Smalltalk a Database System* In *Proceedings of the Sigmod '84 Conference,* June 1984, pp 316-325

[Fujitani 1984] Fujitani, L *Laser Optical Disk· The Coming Revolution in On-Line Storage Communications of the Association of Computing Machinery,* 27, No 6, June 1984, pp 546-554

[Held et al 1975] Held, G D , M Stonebraker and E Wong *INGRES--A relational data base management system Proceedings of the 1975 National Computer Conference,* 44 (1975) pp 409-416

[Hoagland 1985] Hoagland, A *Information Storage Technology A Look at the Future IEEE Computer,* 18, No. 7, July 1985, pp 60-67

[Jones et al 1979] Jones, S , P Mason and R Stamper *LEGOL 2 0 a relational specification language for complex rules Information Systems,* 4, No 4, Nov 1979, pp 293-305

[Katz & Lehman 1984] Katz, R. and T Lehman *Database Support for Versions and Alternatives of Large Design Files IEEE Transactions on Software Engineering,* SE-10, No 2, Mar 1984, pp 191-200

[Lum et al 1984] Lum, V , P Dadam, R Erbe, J Guenauer, P Pistor, G Walch, H Werner and J Woodfill *Designing DBMS Support for the Temporal Dimension* In *Proceedings of the Sigmod '84 Conference,* June 1984, pp 115-130

[Nievergelt & Hinterberger 1984] Nievergelt, J and H Hinterberger *The Grid File An Adaptable, Symmetric Multikey File Structure ACM Transactions on Database Systems,* 9, No 1, Mar 1984, pp 38-71

[Schueler 1977] Schueler, B *Update Reconsidered* In *Architecture and Models in Data Base Management Systems* Ed G M Nijssen-North Holland Publishing Co , 1977

[Snodgrass & Ahn 1985] Snodgrass, R and I Ahn *A Taxonomy of Time in Databases* In *Proceedings of the International Conference on Management of Data,* ACM SIGMod Austin, TX May 1985, pp 236-246

[Snodgrass 1986] Snodgrass, R *A Temporal Query Language ACM Transactions on Database Systems (to appear),* (1986)

[Snodgrass & Ahn 1986] Snodgrass, R and I Ahn *Temporal Databases IEEE Computer (to appear),* (1986)

[Stonebraker et al 1976] Stonebraker, M , E Wong, P Kreps and G Held *The Design and Implementation of INGRES ACM Transactions on Database Systems,* 1, No 3, Sep 1976, pp 189-222

[Svobodova 1981] Svobodova, L *A reliable object-oriented data depository for a distributed computer* In *Proceedings of 8th Symposium on Operating Systems Principles,* Dec 1981, pp 47-58

[Wiederhold 1984] Wiederhold, G *Databases IEEE Computer,* 17, No. 10, Oct. 1984, pp 211-223

[Woodfill et al 1981] Woodfill, J , P Siegal, J Ranstrom, M Meyer and e allman *Ingres Reference Manual* Version 7 ed 1981

Type	Static				Rollback				Historical				Temporal			
Loading	100 %		50 %		100 %		50 %		100 %		50 %		100 %		50 %	
Relation	H	I	H	I	H	I	H	I	H	I	H	I	H	I	H	I
Size, UC= 0	166	115	257	259	129	129	257	259	129	129	257	259	129	129	257	259
Size, UC=14	–	–	–	–	1927	1921	2048	2051	1927	1921	2048	2051	3717	3713	3839	3843
Growth per Update	–	–	–	–	128 4	128 0	127 9	128 0	128 4	128 0	127 9	128 0	256 3	256 0	255 9	256 0
Growth Rate	–	–	–	–	1	1	0 5	0 5	1	1	0 5	0 5	1 99	2	1	1

Notes .
Relation H is a hashed file
Relation I is an ISAM file
'UC' denotes *Update Count*
'–' denotes *not applicable*

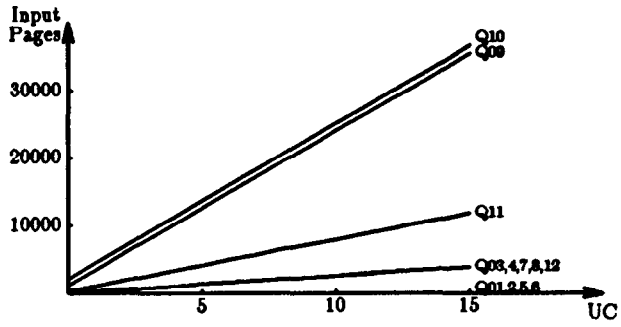
Figure 5: Space Requirements (in Pages)

Update Count	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Q01	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
Q02	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
Q03	129	387	645	903	1153	1411	1669	1927	2177	2435	2693	2951	3201	3459	3717	3975
Q04	128	384	640	896	1152	1408	1664	1920	2176	2432	2688	2944	3200	3456	3712	3968
Q05	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
Q06	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
Q07	129	387	645	903	1153	1411	1669	1927	2177	2435	2693	2951	3201	3459	3717	3975
Q08	128	384	640	896	1152	1408	1664	1920	2176	2432	2688	2944	3200	3456	3712	3968
Q09	1200	3512	5816	8120	10386	12690	14994	17298	19564	21868	24172	26476	28742	31046	33350	35654
Q10	2233	4539	6845	9151	11449	13755	16061	18367	20665	22971	25277	27583	29881	32187	34493	36799
Q11	385	1155	1925	2695	3457	4227	4997	5767	6529	7299	8069	8839	9601	10371	11141	11911
Q12	131	389	647	905	1163	1421	1679	1937	2195	2453	2711	2969	3227	3485	3743	4001

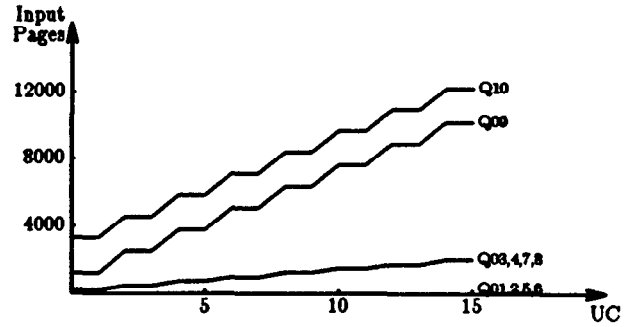
Figure 6: Input Costs for the Temporal Database with 100% Loading

Type	Static		Rollback				Historical				Temporal			
Loading	100 %	50 %	100 %		50 %		100 %		50 %		100 %		50 %	
Query	UC 0	UC 0	UC 0	UC 14	UC 0	UC 14	UC 0	UC 14	UC 0	UC 14	UC 0	UC 14	UC 0	UC 14
Q01	2	1	1	15	1	8	1	15	1	8	1	29	1	15
Q02	2	3	2	16	3	10	2	16	3	10	2	30	3	17
Q03	–	–	129	1927	257	2048	–	–	–	–	129	3717	257	3839
Q04	–	–	128	1920	256	2048	–	–	–	–	128	3712	256	3840
Q05	2	1	1	15	1	8	1	15	1	8	1	29	1	15
Q06	2	3	2	16	3	10	2	16	3	10	2	30	3	17
Q07	166	257	129	1927	257	2048	129	1927	257	2048	129	3717	257	3839
Q08	114	256	128	1920	256	2048	128	1920	256	2048	128	3712	256	3840
Q09	1585	1276	1141	17242	1271	10240	1197	17298	1327	10296	1200	33350	1333	19256
Q10	2214	3329	2177	18311	3329	12288	2233	18367	3385	12344	2233	34493	3385	21303
Q11	–	–	–	–	–	–	–	–	–	–	385	11141	769	11519
Q12	–	–	–	–	–	–	–	–	–	–	131	3743	259	3857

Figure 7: Input Costs for Four Types of Databases



(a) Temporal Database with 100% Loading



(b) Rollback Database with 50% Loading

Figure 8: Graphs for Input Pages

Type	Rollback						Temporal					
Loading	100 %			50 %			100 %			50 %		
Query	Cost (in Pages) Fixed	Cost (in Pages) Variable	Growth Rate	Cost (in Pages) Fixed	Cost (in Pages) Variable	Growth Rate	Cost (in Pages) Fixed	Cost (in Pages) Variable	Growth Rate	Cost (in Pages) Fixed	Cost (in Pages) Variable	Growth Rate
Q01	0	1	1	0	1	0 5	0	1	2	0	1	1
Q02	1	1	1	2	1	0 5	1	1	2	2	1	1
Q03	0	129	1	0	257	0 5	0	129	1 99	0	257	1
Q04	0	128	1	0	256	0 5	0	128	2	0	256	1
Q05	0	1	1	0	1	0 5	0	1	2	0	1	1
Q06	1	1	1	2	1	0 5	1	1	2	2	1	1
Q07	0	129	1	0	257	0 5	0	129	1 99	0	257	1
Q08	0	128	1	0	256	0 5	0	128	2	0	256	1
Q09	0	1141	1 01	0	1271	0 5	56	1144	2 01	56	1277	1
Q10	1024	1153	1	2048	1281	0 5	1080	1153	2	2104	1281	1
Q11	–	–	–	–	–	–	0	385	2	0	769	1
Q12	–	–	–	–	–	–	2	129	2	2	257	1

Figure 9: Fixed Costs, Variable Costs and Growth Rates

Query	Conventional		2-Level Store for Update Count = 14					
	Update Count		Simple	Clustered	Indexed on *amount*			
					as 1-Level		as 2-Level	
	0	14			as Heap	as Hash	as Heap	as Hash
Q01	1	29	–	5	–	–	–	–
Q02	2	30	–	6	–	–	–	–
Q03	129	3717	–	–	–	–	–	–
Q04	128	3712	–	–	–	–	–	–
Q05	1	29	1	–	–	–	–	–
Q06	2	30	2	–	–	–	–	–
Q07	129	3717	129	–	324	30	12	2
Q08	128	3712	128	–	324	30	12	2
Q09	1200	33350	1200	–	–	–	–	–
Q10	2233	34493	2233	–	–	–	–	–
Q11	385	11141	–	–	–	–	–	–
Q12	131	3743	–	–	–	–	–	–

Figure 10: Improvements for the Temporal Database with 100% Loading