LBL-24578

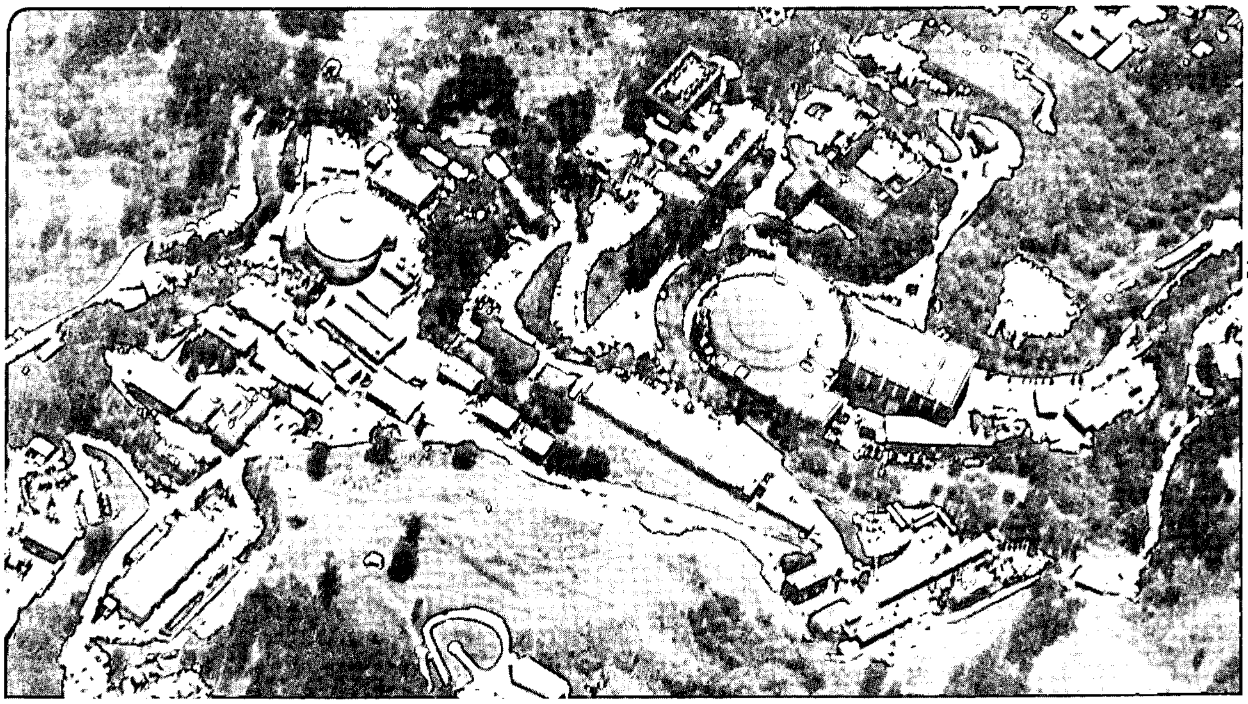
Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Information and Computing Sciences Division

Physical Design of Temporal Databases

H. Gunadhi and A. Segev

June 1988



Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

LBL-24578

Physical Design of Temporal Databases

Himawan Gunadhi and Arie Segev

School of Business Administration
University of California

and

Computing Science Research & Development
Information & Computing Sciences Division
Lawrence Berkeley Laboratory
1 Cyclotron Road
Berkeley, California 94720

June 1988

This work was supported by the Director, Office of Energy Research, Applied Mathematical Sciences Research Program, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

LBL-24578

Physical Design of Temporal Databases

Himawan Gunadhi and Arie Segev

School of Business Administration and
Lawrence Berkeley Lab's Computer Science Research Dept.
University of California
Berkeley, Ca., 94720

Abstract

The primary focus of research into temporal databases has been on logical data modeling. Yet, the feasibility of implementing such databases depends as much on the development of retrieval and storage structures that are suited to the unique nature of temporal data. In this paper, we discuss the major parameters that impact the physical design and distinguish temporal databases from conventional ones. This is followed by a proposal of structures to support the efficient processing of common single-relation temporal queries.

This research was supported by the Applied Mathematics Sciences Research Program of the Office of Energy Research U.S. Department of Energy under contract DE-AC03-76SF00098.

1. Introduction

In the past few years, an increasing amount of attention has been focused on the area of temporal databases (see surveys by [Bolour et al 82; Snodgrass 86; McKenzie 86]). Yet with few exceptions, the focus has been on the design of logical models and query languages. The added dimension of time, and the fact that time itself is not restricted to a single measure [Snodgrass & Ahn 85] implies the need to explore efficient methods of physical storage and retrieval. As [Ahn & Snodgrass 86] observed, building temporal facilities on top of a conventional database degrades its performance significantly.

The majority of literature on logical modeling concentrates on extending the relational model to include and manipulate time. For example, [Snodgrass 87] represented time attributes as {*start* , *end* } pairs appended to each tuple (tuple versioning) and attempted to capture the semantics of time in an extended relational calculus based on Quel. Similarly, [Navathe & Ahmed 87; Ariav 86] introduced an extended SQL under tuple-versioning. On the other hand, [Clifford & Tansel 85] proposed attribute-versioning, whereby each temporal attribute is individually indexed by time. A completely different perspective, based on the concept of the *Time Sequence Collection* (TSC), has been adopted by [Shoshani & Kawagoe 86; Segev & Shoshani 87].

In the area of physical modeling, [Lum et al 84] proposed a scheme based on reverse chaining of historical data: index-trees are maintained on non-time attributes, and one or two additional levels of indirection are needed in order to retrieve specific time values. [Stonebraker 86] proposed forward chaining of historical data without maintaining any form of indexing on time itself. The first tuple of a given relation is stored as an anchor record in its entirety, while each subsequent tuple merely captures changes and is chained to its immediate predecessor. These techniques are effective only where queries can be decomposed into those that retrieve complete (or most of the) histories of single entities.

On the other hand, [Rubenstein 85] discussed indexing methods for time-ordered data. These are limited to information that is inherently time-order sensitive, such as musical notes or production scheduling, and are not applicable to the broader notion of temporal databases. In [Rotem & Segev 87], partitioning techniques to store temporal data along the time and surrogate† dimensions were explored.

In this paper, we first discuss (Section 2) the major parameters that impact the physical design; these include the way data and time are represented (which requires basic interpolation capabilities), the life-span of the data, and the type of queries. We then propose, in Section 3, indexing structures to support the efficient processing of common single-relation queries. These structures are evaluated in terms of storage and retrieval costs, and are compared with previous studies. Section 4 discusses the ways that hybrids of the cases in Section 3 can be handled. Section 5 concludes the paper with a summary and directions for future research.

2. Processing Requirements for Temporal Data

There are several factors that have to be explicitly considered in the physical design of temporal databases. In the following we shall present them and evaluate their impact on design.

Triplet Representation of Temporal Data

Temporal data can be represented as a triplet of (S, T, A) [Segev & Shoshani 87], where S = surrogate, T = time and A = temporal attribute. Figure 1 illustrates a simple example of such a *Time Sequence Collection* (TSC), where S = employee ID, T = month, and A = salary in thousands. For each surrogate, there exists one or more $\{time, value\}$ pairs. We might have represented the *TSC* in various other ways, e.g. as a list of the form:

$$\{(100, (Jan, 20), (Mar, 25)\ (Dec, 30)), (101, (Jan, 30), (Mar, 40), (Dec, 50))\}$$

We can generalize this conceptual representation to account for the presence of multiple surrogates, time lines, and attributes. Multiple surrogates is the equivalent of composite keys, thus for indexing purposes we can treat it as a single entity. In the case of temporal-attributes, the usefulness of the preceding definition of temporal data is that the set of attributes always form a

† For our purposes, we use the term surrogate to describe time-invariant keys, i.e. the primary key of the snapshot version of a temporal relation. In general, only a subset of candidate keys in a relation would qualify as surrogates.

	Jan	Mar	Dec
100	20	25	30
101	30	40	50

Table 1. Example of a Salary *TSC*

synchronous set as defined by [Navathe & Ahmed 87]. We concentrate here on the case of a single attribute and a single time line, although the analysis can be easily extended to handle the presence of multiple attributes. We will also assume a relational representation of the data.

Representation of the Time Attribute

As we are interested in retaining the basic relational representation of data, attribute-versioning models will be ignored, since they require the maintenance of non-first normal form (-1NF) relations. There are two types of tuple-versioning that we shall consider: interval and event (time) stamping. Using the example given in Figure 1, Figure 2 shows a relational representation with interval-stamping, while Figure 3 shows the event-stamped version.

ID	SAL	T_S	T_E
100	20	Jan	Feb
100	25	Mar	Nov
100	30	Dec	Dec
101	30	Jan	Feb
101	40	Mar	Nov
101	50	Dec	Dec

Table 2. Interval-Stamped Temporal Relation

ID	SAL	T
100	20	Jan
100	25	Mar
100	30	Dec
101	30	Jan
101	40	Mar
101	50	Dec

Table 3. Event-Stamped Temporal Relation

The disadvantage of interval-stamping is that extra storage is needed for the additional time-stamp. On the other hand it has two advantages: first, an interpolation can be made without reference to an adjacent tuple (either a predecessor or successor); and second, discontinuities within the life-span of a temporal attribute need not be explicitly represented. Interpolation is a unique and critical aspect of temporal data processing, since there is unlikely to be a one-to-one correspondence between valid time points and recorded data points (see [Segev & Shoshani 87] for details). Discontinuities in life-spans, on the other hand, impacts the correctness of interpolation.

The Life-span and Operations on the Relation

The life-span of a temporal relation predetermines the types of storage and retrieval strategies that can be developed. A *static* life-span corresponds to a set of data that is completely historical, i.e. no appends will be made to it. This is relevant in such areas as scientific, decision support and economic/econometric data analysis. In such cases, we need only concern ourselves with rapid retrieval of data, and ignore the possibility of future reorganization of the database. This enables for example, the pinning of records.

A *dynamic* life-span requires on the other hand the continuing expansion of the database. Such a database may be append-only (e.g.Postgres [Stonebraker & Rowe 85]), or one that also allows deletion and replacement. The direction in temporal database research is towards append-only databases, especially in the light of recent advances in optical disk technology. Dynamic temporal databases provide many challenges, including that of manageability of the indexes [Lum et al 84], memory hierarchy management [Stonebraker 87b] and conceptual problems in inferencing [Navathe & Ahmed 87b]. In this paper, we focus on static life-spans.

Nature of Queries

For the purposes of physical requirements, queries can be divided into those that require a single relation and those that need joins - both *thetha* joins and those based on time. We will illustrate the queries with examples.

Surrogate and/or Attribute: these can be expressed as $S = s$, $A = a$, and conjunctive forms involving them. Disjunctive forms, the use of non-equality signs and range specification over S or A can essentially be converted to equivalent forms using the above. These queries assume a default time specification of $T = ALL$. With reference to the previous Tables 1 to 3, examples are:

Q1. "What is the salary history of employee with ID = 100"

Q2. "Find the employees that earn more than 20K"

Time: there are several ways in which time can be specified by itself. It may be $T = t$, i.e. a specific point, $T = [t_i, t_j]$, i.e. over an interval, and also over a set of intervals (periods). These do not specify surrogate or attribute values. Further, time related predicates may also require interpolation.

Q3. "Which employees joined the company in February"

Q4. "Which employees worked for the company for at least two years"

Surrogate/Attribute Qualified on Time: These involve conjunctive forms of the above two. Disjunctive forms can be decomposed into independent queries and executed separately.

Q5. "Find the salary of employee ID = 100 between January and June"

Temporal Ordering: These are similar to the above except that ordering is required for one or more time points. Ordering may be anchored on the beginning, end or a specified time point along the life-span of the relation.

Q6. "What were the last two salaries of employee ID = 100"

Q7. "What was the second salary of employee ID = 101 between the months of February and September"

Aggregates: unlike conventional relations, the semantics for temporal aggregates is much more complex and yet richer. Aggregates can be derived along the time or surrogate dimensions, and they can also be derived by accumulation along time [Segev & Shoshani 87].

Q8. "What was the monthly wage bill for the first two years"

For an example of accumulation, consider the event-stamped relation of Table 4.

TYPE	SALES	T
Business	20	Jan
Business	25	Feb
Business	30	Mar
CS	30	Jan
CS	40	Feb
CS	50	Apr

Table 4. Event-Stamped Relation For Book-Sales

Q9. "Find the cumulative sales of Business books for the past year"

Joins: Joins may require the evaluation of binary temporal predicates that have as operands time values, ranges, periods or ordering from each of the participating relation. Unlike conventional joins, the time-attribute may or may not be used as the joining attribute.

3. Structures for Static Single Relation Processing

With the foregoing discussion in mind, we start by studying the most fundamental design issues. In the context of the given framework, our emphasis in on (1) a single time-line, (2) both event and interval stamped tuple-versioning, (3) static databases, and (4) single relation processing.

Access requirements on single relations can be dichotomized along the primary qualification of the queries, namely (1) surrogate, (2) attribute, and (3) time. Our approach is to initially look into each case separately, attempting to optimize the access and storage structures to satisfy the primary query qualification. We then examine the support for hybrid qualifications.

In order to be able to put any performance analysis into perspective, comparisons will be made to those suggested in [Lum et al 84] and [Rowe & Stonebraker 87; Stonebraker 86], which we will refer to as Lum and Postgres† respectively. An important assumption of this paper is that we are dealing with very large databases, and as such are not concerned with trivial length histories. This influences our approach to the design of the structures, which will not be efficient

† We compare our structures to Postgres for the sake of completeness. However, it should be noted that Postgres has not been designed to provide generalized temporal support, and its temporal structures are primarily for the replacement of the traditional backup & recovery methods.

both in retrieval and storage overheads when compared to the other two approaches if small databases are included. A second assumption is that we are dealing primarily with temporal sequences that are step-wise constant, that is, the attribute value at a time point remains constant until the next explicit value. Sequences of discrete type can be treated as a special case of step-wise constant sequences. A discussion of sequence types can be found in [Segev & Shoshani 87]. We will also continually emphasize the importance of making correct and efficient interpolations, and how this influences the design.

3.1. Surrogate Indexing

Motivation

In a snapshot or time-sliced version of a temporal relation, the surrogate acts as the primary key. It is natural to expect a great number of queries to be qualified primarily on this domain. The unique identifier of the complete relation though is a composite of the surrogate and one time stamp (either the start or end time in the case of interval-stamped relations). Dense indexing via the composite key may be too expensive for large relations, since the size of the index will be of the order of $O(N)$, where N is the cardinality of the relation. Not only will storage cost be high, but so will the cost of retrieving a data tuple, due to the height of the resulting tree. If instead, a sparse index on the composite key were constructed, then retrievals based on specific surrogate values cannot be efficiently made. Hashing on the surrogate is impractical here because of the long histories. In addition hashing on either the surrogate or the surrogate/time identifier will cause total loss of the temporal ordering.

Description

We resolve the above dilemma by developing a method that employs a two-level index as shown in Figure 1. A first level dense index is provided on the surrogate values (the surrogate index), and a second level sparse index is constructed over the life-span of each surrogate value (the time-index). We employ a B^+-tree for surrogate indexing, and an indexed sequential access method (ISAM) organization for the time-index. The choice of ISAM for the time-index is due to

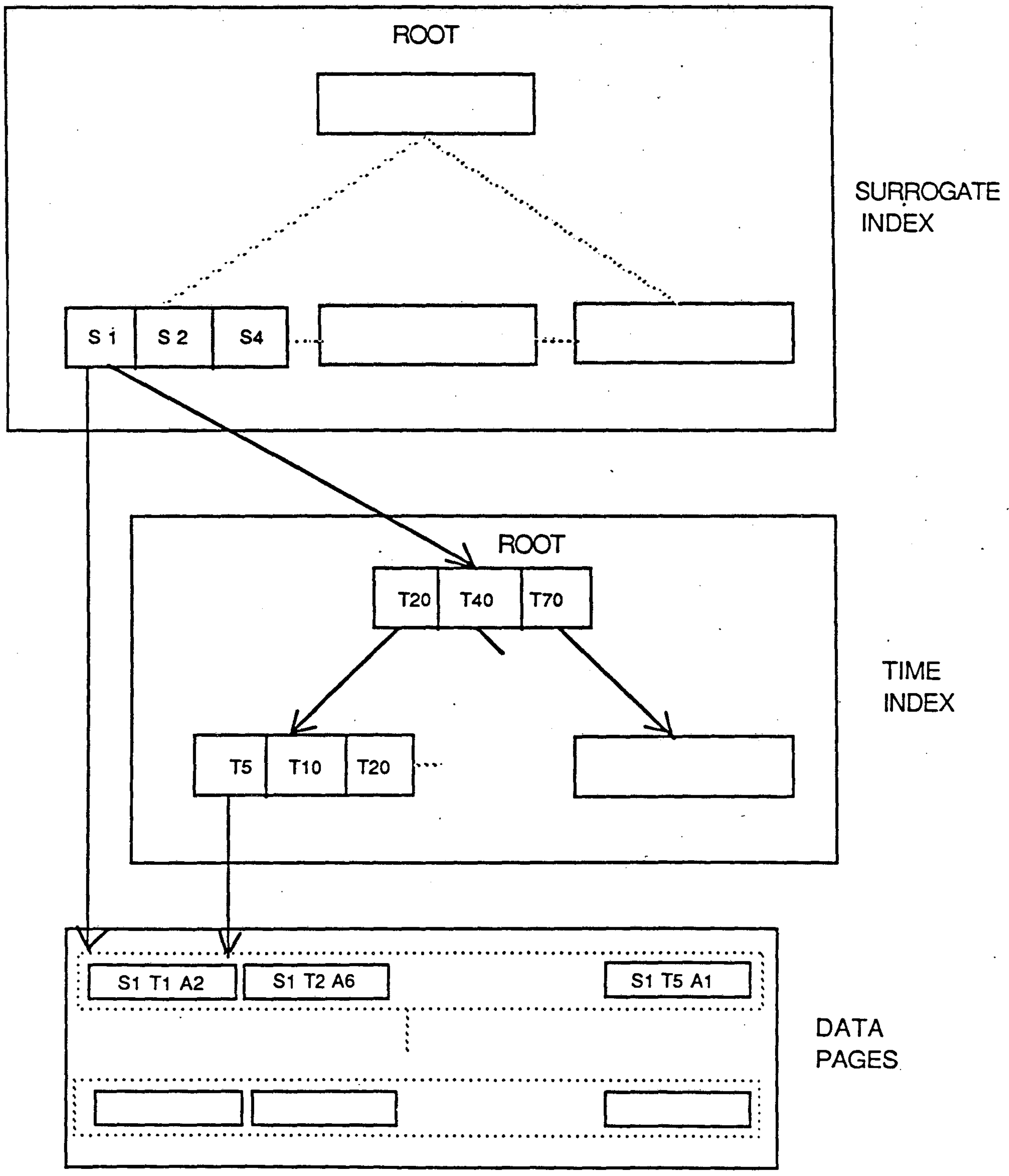


FIGURE 1. SURROGATE INDEXING

the need to minimize the index size, and also the observation that no overlaps exists among the time-stamps of the tuples associated with each surrogate, thus yielding a chronological ordering of data.

In the leaves of the B^+-tree, each index-key value has associated with it not the appropriate tuple identifier as in conventional databases, but a pointer to the root of the associated time-index. The only modification that we have made to the B^+-tree is the addition of a second pointer field for each leaf-entry. This pointer leads directly to the first data page that stores the data tuples of the surrogate. When a query requests the complete history of a surrogate, the second pointer enables the time-index to be bypassed. If instead, the information were stored in the root of the time-index, this would force another index page to be retrieved before the first data page address is obtained.

The ISAM organization of the time-index requires physical sorting of data tuples along the time range for each surrogate, but this does not cause any problems for static lifespans. At the leaf-level of the index, the pointer associated with an index-key value points to the disk page that contains tuples with time-stamps that are less or equal to it, but greater than the preceding index-key value. For interval-stamped relations, either the start or ending stamp may be used to determine key values, but we shall assume in our discussions that the starting time-stamp is selected.

An important issue in constructing the index is its ability to aid interpolations along the time-line. The sparsity of the index does not result in redundant retrieval of data pages when compared to dense indexing, due to the chronological ordering of data. The issue that primarily influences the performance of interpolation is the treatment of discontinuities in a surrogate's life-span. There are two ways in which discontinuities can be represented within the index. The first omits any reference to it within the index, which means that only when the relevant data page or pages are read, will the discontinuity be discovered. The second technique in contrast explicitly represents null values within the life-span by inserting a special marker into the appropriate *leaf* of the index. This is illustrated in Figure 2, where such an index is constructed

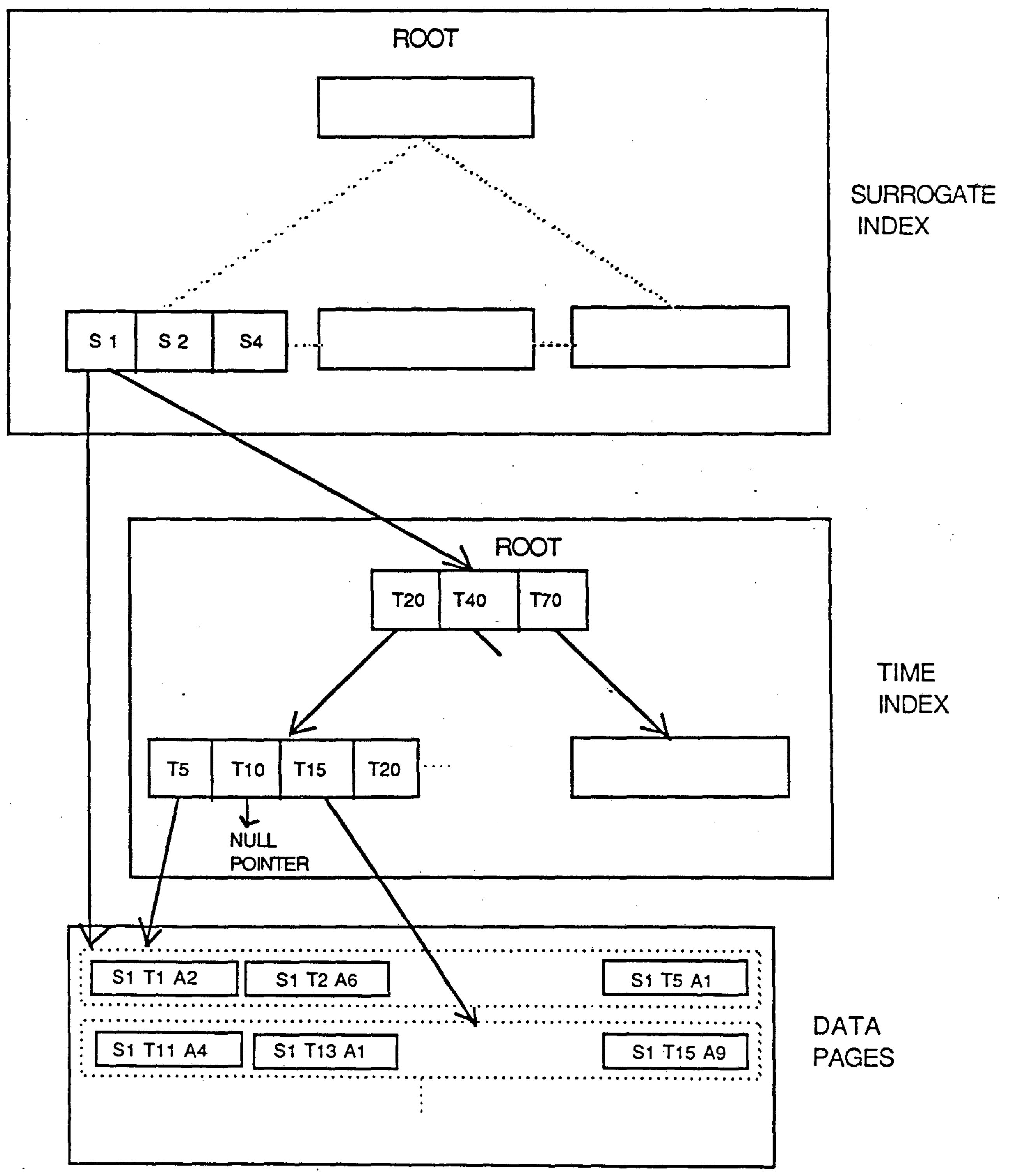


FIGURE 2. REPRESENTATION OF A DISJOINT LIFE-SPAN IN THE TIME-INDEX

for a surrogate lifespan that is disjoint.

We have selected a null representation that does not require a record of different length or type from those utilized for the key values. The only difference is that the associated pointer is set to null. The necessary intelligence can be built into the search algorithm to interpret this fact. Explicit consideration of discontinuities in the index ensures that data is not accessed redundantly, but at the expense of placing a constraint on the way data tuples are partitioned by the index. This restriction arises from the fact that the null indicator takes the place of a key value that would have represented a page of data. The effect on the index size of a null indicator is thus equivalent to adding B_d tuples to the relation, where B_d is the blocking factor for data tuples. Which method of treating breaks in the life-span is preferable depends on their frequency in the relation in question.

Performance Analysis

The cost of storing the surrogate-index is $O(\mid S \mid)$, where $\mid S \mid$ is the cardinality of the surrogate domain on the relation. The storage cost of a time-index is $O\left(\frac{N}{\mid S \mid B_d}\right)$, where $N / \mid S \mid$ is the average length of the history chain pertaining to a surrogate. Since each surrogate maintains its own time-index, the total cost of the time-indexes for a relation will be approximately $O\left(N / B_d\right)$. This two-level access structure may mean considerable savings in storage space relative to dense indexing or Lum's approach, both of which require space that is proportional to the size of the relation.

If the maximum number of children a node can have in the surrogate and time trees are k_1 and k_2 respectively, then the cost of retrieving any surrogate value is $O\left(\log_{k_1} \mid S \mid\right)$, which is independent of the length of the history chain. The cost of retrieving a specific time value for a surrogate is $O\left(\log_{k_2} \frac{N}{\mid S \mid B_d}\right)$, and is unlikely to exceed 3 disk accesses. As an example, if the average length of history is 100,000 tuples, and B_d is 20, then there are 5,000 keys to be stored in the time-index pages. If we assume that each disk page can hold 100 key-pointer records (say a 1,024 byte page with 10 byte records made up of a 4 byte pointer and 6 byte time-stamp fields), then we need just 50 leaf-level pages for the index, thereby producing a two level tree. In fact a three-level index will be capable of storing information pertaining to some 20 million tuples, if we maintain the above assumptions.

Queries that request whole histories of a surrogate require a B^+-tree access plus sequential retrieval of the data blocks (recall that the history of each surrogate is contiguous and physically ordered by time). If a specific time qualification is given, the cost is equal to the height of the B^+-tree plus the height of the time-index for the surrogate value plus one page retrieval to obtain the tuple itself. If time is specified over a range, the cost is of the same order as a single time-value qualification, since data is sorted along the time dimension. On the other hand, queries that request a range of surrogates increase the cost of time-index and data page retrievals by a factor equal to the number of qualifying surrogates, while the surrogate index itself needs to be scanned just once.

Lum's method would fare worse for queries not involving whole histories, since he proposed the use of pointer-chains or a linear list to store the {time-stamp, data-pointer} pairs associated with a surrogate, the surrogates themselves being indexed by a B^+-tree. This implies that the cost of retrieving a given time-value is $O\left(\frac{N}{|S|}\right)$, i.e. proportional to the size of the history itself. Postgres' scheme can only provide equivalent performance to our proposal for the retrieval of complete histories, since no index is kept on time, and the tuples with a given surrogate value are chained into a linear-list. Thus the cost of queries qualified on time will be of the order $O(N)$.

3.2. Attribute Indexing

Motivation

Relational attributes other than the surrogate are inherently multi-valued. In the case of temporal relations, there is a many-to-many relationship between a temporal attribute and the surrogate and time domains. Furthermore, the time-intervals ranging over a given attribute value do not naturally fall into a disjoint set. In other words, there are likely to be many overlaps among the time-intervals found in tuples with the specific attribute value.

This problem was ignored by Lum, who considered only the multiplicity of surrogates for a given attribute value, apparently assuming that associated time intervals can be ordered in the same manner as for the case of surrogate-indexing. Postgres on the other hand is not designed to answer arbitrary queries involving historical values of an attribute, since no indexed access is available to non-current values.

Description

We develop an access method that has three levels of indirection (see Figure 3): the first of which indexes the attribute values (attribute-index), the second indexes the time values for each attribute (time-index), and the last level made up of one or more disk pages containing pointers to data for pairs of {attribute, time} values (pointer-records). The additional level of indirection over the surrogate indexing previously introduced is needed due to multiplicity of values and the desire to keep the time-index manageable in size.

The attribute-index is a B^+-tree ranging over the values found in the relation. Within the leaf-level pages, each record maintains two pointers per key value. The first pointer leads directly to the pointer-records, in order to answer queries not further qualified on time. The second pointer leads to the the time-index for that attribute value. The time-index is structured also as a B^+-tree, and each index entry represents an interval boundary. Due to the fact that we do not assume any sort order of physical data along the temporal attribute, sparse indexing along the lines of an ISAM organization is not feasible, which is why we have selected the B^+-tree. Yet we avoid having to index the data according to actual time intervals. This will be further elaborated in the next section. On the last level, each pointer-record is of variable length, in that it may take more than one disk page if need be. These pointer records are sorted or at least clustered according to the relevant attribute and time intervals, so that sequential retrieval can be accomplished.

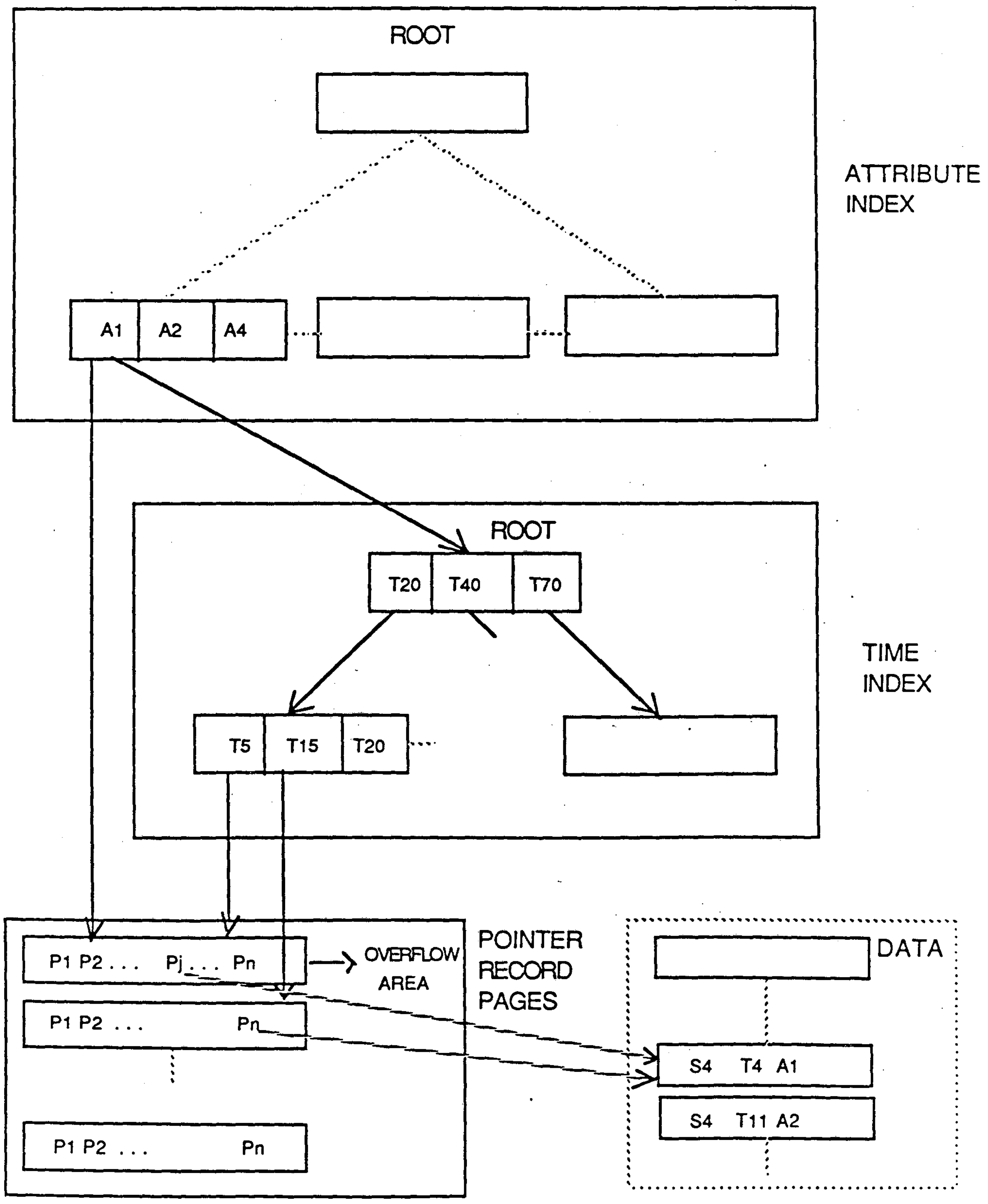


FIGURE 3. ATTRIBUTE INDEXING.

Partitioning the Time Intervals

Let a_i be a specific occurrence of the temporal attribute A, with cardinality of N_{a_i}. Associated with a_i is a set of data tuples with time intervals, $I_1, I_2, \ldots, I_n$, where $t_s(I_j)$ and $t_e(I_j)$ represent the starting and ending points of the interval I_j. In general, it is unlikely that the intervals are disjoint, nor will they have uniform spans. If we let $k_1, k_2, \ldots, k_m$ denote the leaf entries of the time-index, it is likely that there exists an I_j that covers or overlaps with an index interval bounded by $[k_l, k_{l+1}]$.

Thus if the k_l s are not judiciously selected, the end result may have a high degree of imbalance in terms of the number of pointers associated with each time interval of the index. The balancing of the pointer assignments amounts to trying to minimize the worst-case retrieval cost. Note that data pointers can be duplicated in more than one pointer record accessed from the leaves of the time index. The approach we develop first determines the k_l values, assuming that no duplication occurs. Then after the assignment of I_j s has been completed, we attempt to reduce the overflows resulting from duplication by adjusting the boundaries of the leaf index entries. Once the partitioning algorithm has terminated, we can construct the time-index tree bottom-up.

Algorithm Partition

[Step 1] Call Initial-Assign

[Step 2] Call Balance

Algorithm Initial-Assign

[Step 1] Let N_k be the number of leaf nodes in the index tree and B_p the number of pointers that a pointer record can hold.

$$N_k = \frac{N_{a_i}}{B_p}$$

[Step 2] Let L be the minimum binding interval for a_i, i.e. $L = t_{N_k} - t_0$, where $t_0 = \min_j\{t_s(I_j)\}$ and $t_{N_k} = \max_j\{t_e(I_j)\}$.

Set the index key entries

$$k_j = j * \frac{L}{N_k}, \text{ for } j = 1, \ldots, N_{k-1}$$

[Step 3] Let N_{k_j} be the number of pointers assigned to the pointer record indexed by k_j. Assignment of I_j to such a record depends on the intersection between it and the index intervals.

$$N_{k_j} = \sum_{l=1}^{N_{a_t}} X_l$$

$$\text{where } X_l = \begin{cases} 1 & \text{if } I_l \bigcap (k_{j-1}, k_j] \neq \emptyset \\ 0 & 0 \text{ otherwise} \end{cases}$$

Algorithm Balance

[Step 1]

If $N_{k_j} = B_p \cup N_{k_j} = n$, $\forall j$, EXIT.

Else let $N_{k_{\max}} = \max_j \{N_{k_j}\}$.

[Step 2] If $N_{k_{\max}} = N_{k_{max-1}} \cap N_{k_{\max}} = N_{k_{max+1}}$ then call Algorithm Balance with index key set $= \{k_l\} - k_{\max}$.

Else

(1) if only $N_{k_{\max}} < N_{k_{max+1}}$, find an interval I_j such that $\min_l \{k_{\max} - t_s(I_l)\}$ holds true. In case of a tie, select the interval such that $t_e(I_j) > k_{\max}$. Otherwise pick one.

(2) if only $N_{k_{\max}} < N_{k_{max-1}}$, same procedure as (1), but applied to the left boundary.

(3) when both boundaries apply, consider (1) and (2) in conjunction. But for the tie-breaking rules, when all or none of the candidate I_js for removal are already indexed by one of the neighboring intervals, then find $\min\{N_{k_{max-1}}, N_{k_{max+1}}\}$, and select the boundary associated with the minimum. Otherwise pick one.

[Step 3] Move the appropriate boundary of $k_{\max}$ inwards such that I_j can be discarded from the pointer record associated with $k_{\max}$.

Decrement $N_{k_{\max}}$ and increment that of the neighbor if necessary.

Go to Step 2.

Performance Analysis

The partitioning algorithm presented is a heuristic one, and can be further improved in the following way. After the termination of the algorithm, If the maximum number of pointer blocks corresponding to some time-index interval is unacceptable, the number of leaf entries (m) can be increased and the balancing procedure repeated. There are several improvements that are possible. Alternatively, if P is the number of pages per record, we can then call the partitioning algorithm again with the parameter B_p modified to $B_p * \frac{1}{P}$.

The storage cost of the indexing structures for the temporal attribute is made up of the cost of the attribute index, time index and the pointer records. The size of the attribute index is of the order $O(\mid A \mid)$, i.e. proportional to the cardinality of the attribute domain of the relation. The exact size of the history chain for a given instance of the attribute is data dependent, but we can "overestimate" the mean value to be $\frac{N}{\mid A \mid}$. Since the time index proposed above is non-dense, the storage required for the domain of the attribute is of the order $O(\frac{N}{B_p})$. Although a large number of pointers may be duplicated in the pointer records, the blocking factor B_p would also mean that the resulting tree will be relatively short. The pointer records themselves have a storage cost of $O(N)$, but it should be kept in mind that *only* pointers are maintained there, which normally take up four bytes of memory each.

The main difference in cost between the access structures for the temporal attribute and the surrogate is the additional disk page that must always be retrieved before data pointers are obtained. The orders of magnitude of the time needed to search the multi-level index in order to respond to queries on a given attribute value, range of values, conjunctive time qualification over both single points and intervals are similar to those derived for the surrogate structures. The time needed to retrieve the tuples from physical storage will be linear to the number of pointers in the pointer record retrieved. Since we do not maintain time stamps alongside the data pointers, information may not be adequate in answering some queries. Likewise, certain types of interpolation cannot be made without searching through the whole pointer record. Yet we do not include the time-stamps with the data pointers because we would have to include both the starting and ending times (even for event stamped relations) and this will likely result in high overheads.

As pointed out before, Lum's proposed structures cannot effectively answer which queries due to the treatment of the time intervals. The Postgres approach is also not applicable since it was not planned to retrieve past histories of secondary keys.

3.3. Time Indexing

Motivation

An important type of query neglected by the previous structures is that which is qualified solely on time, for example, "what were the salary values in December?" More than in the design of the previous two structures, efficient indexing of the time dimension is greatly dependent on the physical organization of data. We see the solution as twofold, one that requires primary data sorting on time values which will require changes to the surrogate access structures previously explained, or one that is based on the time index for attributes.

Description

The first method † of time indexing is based on physical organization of the data along the time dimension. Since queries qualified solely on time are mostly range queries or otherwise require interpolation, having tuples sorted according to the time attribute and constructing a sparse index above it will provide for efficient query processing. The construction of such an index for static databases is straightforward. We partition the lifespan into n segments, and construct a B-tree bottom-up. The leaf index entries will have to store only the page address or the first such page if more than one qualifies. If a segment corresponds to a single disk block, then we might have a tall tree to access. On the other hand, if a variable number of blocks are chained together

† We skip a detailed description of the structures since they are simple variations of the preceding ones.

in order to reduce the height of he tree, then the speed with which tuples are retrieved will be slower, i.e. the index is sufficiently discriminating. Note that we need maintain sortedness only amongst disk pages.

The alternative indexing method does not require a specific organization of the data. What we do is adapt the second and third levels of the attribute structures of the previous subsection to act as a dense index on time. The tree itself can again be controlled in terms of its height, but we will need to make random disk reads given a set of data pointers pointed to by an index entry. The performance of two indexes are similar to their counterparts for the previous two indexes.

4. Integration of Hybrid Qualifications

In the last section we looked at access and storage methods that are focused primarily on queries primarily qualified on the surrogate, temporal attribute or time. Here we will briefly look into how we can integrate the different structures in order to respond to a broader range of queries. The methods presented in Section 3 are not designed to handle the the following cases: (1) they cannot respond to queries qualified on attribute and surrogate values and (2) do not adequately handle range queries on either the surrogate or attributes.

One possibility is to use the structures that we developed earlier and make appropriate modifications where appropriate. For example we found that we could maintain a smaller time index in terms of storage and search space if we can sort the data along this dimension. Surrogate indexing required temporal sorting of physical data for each surrogate. We can resolve this by looking at the trade-offs between the additional cost of surrogate qualifications in the case of primary time sorting and the additional cost of time qualifications in the case of primary surrogate sorting. Queries that are qualified on both surrogate and temporal attribute can be resolved by decomposing them into two independent single qualification subqueries, retrieve the data pointers from the separate access structures, and finally find the intersection between the two sets.

Another approach is to employ multidimensional partitioning of the data file. In [Rotem & Segev 87] symmetric and asymmetric algorithms for the case of static data were presented. Such techniques may provide a good solution to queries with conjunctive qualification on the temporal attribute and surrogate, because we can now find a physical ordering of the tuples along the surrogate and temporal attribute. Furthermore, range qualifications can be more efficiently satisfied since data is partitioned into intervals along each dimension. The access structures can be constructed sparsely over the data for each indexing key. Thus the index sizes could be kept relatively small. One research issue related to this approach is how overall performance degenerates with respect to the number of partitioning attributes and the distribution functions of these attributes.

5. Summary and Future Research

In this paper, we have discussed the major parameters that effect the physical design of temporal databases. These parameters include the way data and time are represented, the nature of the life-span, and the type of queries. We have then proposed indexing structures for the case of relational representation of the data, single-relation retrievals, and static life-spans.

The main properties which are unique to temporal databases are:

1) Temporal ordering - unlike the relational model, it is necessary to maintain temporal ordering of the data for a given surrogate. In fact, there are applications (such as scientific experiments) where the 'current' version is meaningless.

2) Interpolation - in many applications, viewing the temporal data as a collection of time sequences enables interpolation. This is very important in environments with very large data historical collections (Giga and Tera bytes). Incorporating interpolation can be viewed as a form of data compression. The simplest (and probably the most prevalent) case is the stepwise-constant sequence that describes state variables. In this case, the value of a variable (e.g., salary) remains the same until the next recorded change.

3) Discontinuities in histories - there are time sequences which are discontinuous; for example, if an employee leaves a company and later rehired, the salary history will be discontinuous.

The above properties are interrelated. For example, interpolation requires temporal ordering, and should handle discontinuities. The contribution of this paper is the presentation of structures to support the above properties, and a comparison with other proposals. The problem of time indexing is not a simple one; one has to decide on the number of sub-intervals and their specific time boundaries. We have proposed an algorithm to do that with the objective of balancing the number of pointers across the sub-intervals. By doing so, we are trying to minimize the worst-case retrieval time. It should be noted, however, that this problem can be approached with other objectives and algorithms (which are currently being studied).

This work represents one of the initial explorations of physical design of temporal databases. Current and future work deals with extending the investigation into cases of multiple time lines, dynamic append-only databases, and multiple-relation processing. We are also looking at possible non-relational physical representations.

References

[Ahn 86] Ahn, I., Towards an Implementation of Database Management Systems with Temporal Support, *Proceedings of the International Conference on Data Engineering*, 1986, pp. 374-381.

[Ahn & Snodgrass 86] Ahn, I., Snodgrass, R., Performance Evaluation of a Temporal Database Management System, *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1986, pp. 96-107.

[Ariav 86] Ariav, G., A Temporally Oriented Data Model, *ACM Transactions on Database Systems*, 11, 4, December 1986, pp. 499-527.

[Bolour et al 82] Bolour, A., Anderson, T.L., Dekeyser, L.J., Wong, H.K.T., The Role of Time in Information Processing: A Survey, *ACM-SIGMOD Record*, 12, 3, 1982.

[Clifford & Tansel 85] Clifford, J., Tansel, A., On an Algebra for Historical Relational Databases: Two Views, *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1985, pp. 247-265.

[Comer 79] Comer, D., The Ubiquitous B-Tree, *Computing Surveys*, 11, 2, 1979, pp. 121-138.

[Guttman 84] Guttman, A., R-trees: A Dynamic Index Structure for Spatial Searching, *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 1984, pp. 47-57.

[Lum et al 84] Lum, V., Dadam, P., Erbe, R., Guenauer, J., Pistor, P., Walch, G., Werner, H., Woodfill, J., Designing DBMS Support for the Temporal Dimension, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1984, pp. 115-130.

[Navathe & Ahmed 87] Navathe, S.B., Ahmed, R., TSQL- A Language Interface for History Data Bases, *Proceedings of Temporal Aspects of Information Systems*, North-Holland, May 1987, pp. 113-128.

[Rotem & Segev 87] Rotem, D., Segev, A., Physical Organization of Temporal Data, *Proceedings of the International Conference on Data Engineering*, February 1987, pp. 547-553.

[Rubenstein 85] Rubenstein, W.B., Indices for Time-Ordered Data, Masters Thesis, Computer Science Division, University of California, Berkeley, CA, May 1985.

[Segev & Shoshani 87] Segev, A., Shoshani, A., Logical Modeling of Temporal Databases, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1987, pp. 454-466.

[Shoshani & Kawagoe 86] Shoshani, A., Kawagoe, K., Temporal Data Management, *Proceedings of the International Conference on Very Large Data Bases*, August 1986, pp. 79-88.

[Snodgrass 87] Snodgrass, R., The Temporal Query Language TQuel, *ACM Transactions on Database Systems*, June 1987, pp. 247-298.

[Snodgrass 86] Snodgrass, R., Ed., Research Concerning Time in Databases Project Summaries, *ACM-SIGMOD Record*, 15, 4, December 1986.

[Snodgrass & Ahn 85] Snodgrass, R., Ahn, I., A Taxonomy of Time in Databases, *Proceedings of ACM SIGMOD International Conference on Management of Data* , May 1985, pp. 236-246.

[Stonebraker 86] Stonebraker, M., Inclusion of New Types in Relational Data Base Systems, *Proceedings of the International Conference on Data Engineering*, February 1986, pp. 262-269.

[Stonebraker 87] Stonebraker, M., The Design of the Postgres Storage System, *Proceedings of the International Conference on Very Large Data Bases*, August 1987, pp. 289-300.

[Rowe & Stonebraker 87] Rowe, L.A., Stonebraker, M.R., The POSTGRES Data Model, *Proceedings of the International Conference on Very Large Data Bases*, August 1987, pp. 83-96.