Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

QUERY PROCESSING AND OPTIMIZATION IN TEMPORAL DATABASE SYSTEMS

T.-Y. Leung

June 1992
CSD-920036

University of California
Los Angeles

Query Processing and Optimization in Temporal Database Systems

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Ting Yu Leung

1992

The dissertation of Ting Yu Leung is approved.

[illegible]

Kirby Baker

Jack Carlyle

Jack Carlyle

[illegible]

Michel Melkanoff

[illegible]

Siu Tang

[illegible]

Richard Muntz, Committee Chair

University of California, Los Angeles

1992

Table of Contents

List of Figures

List of Tables

Acknowledgments

First of all, I would like to express my sincere thanks to my dissertation advisor, Richard Muntz, for his guidance, encouragement, and friendship. Through his patient guidance, I overcame many difficulties during my studies at UCLA, and through his insightful criticism, I produced quality work. I also thank other members of my committee, Kirby Baker, Jack Carlyle, Michel Melkanoff, and Siu (Chris) Tang, for their time and suggestions.

Additionally, I thank my fellows at UCLA for their invaluable discussions, comments, and friendship: William Cheng, Leana Golubchik, and Steven Berson. Special thanks go to Verra Morgan who gave me plenty of advice and courage on parenting. I will never forget her spontaneous response when I said to her "Tell you a good news!".

I am thankful to my parents, Kwok and Shun-Yeung, for their constant encouragement of getting higher education, and to my daughter, Karyan, for the joy that she brought to this world. I am also grateful to other family members, particularly Wendy and Kin, for their support throughout the course of my graduate studies. Last but not least, I am indebted to my wife, Agnes, for her endless love, patience, and devotion. I dedicate this dissertation to her.

This work was partially supported by DARPA contract F29601-87-C-0072 (Tangram project), and a MICRO grant from the University of California and the Hughes Aircraft Company.

Vita

1962	Born, Hong Kong.
1984	B.S. Electronics. The Chinese University of Hong Kong.
1985–1986	Teaching/Research Assistant, Computer Sciences Department, The University of Texas at Austin.
1986	M.S. Computer Science. The University of Texas at Austin.
1986–1989	Teaching/Research Assistant, Computer Sciences Department, UCLA.
1990	Senior Member of Technical Staff, Teradata Advanced Concepts Laboratory, Teradata Corporation.
1991	Programmer Analyst, Medical Imaging Division, Radiological Sciences Department, UCLA.
1991–present	Research Assistant, Computer Science Department, UCLA.

Publications

T.Y. Leung and R.R. Muntz, "Generalized Data Stream Indexing and Temporal Query Processing," *Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing (RIDE-TQP)*, February 1992.

T.Y. Leung and R.R. Muntz, "Query Processing for Temporal Databases," *IEEE Sixth International Conference on Data Engineering*, February 1990.

D.S. Batory, T.Y. Leung, and T.E. Wise, "Implementation Concepts for an Extensible Data Model and Data Language," *ACM Transactions on Database Systems*, September 1988.

T.Y. Leung and R.R. Muntz, "Temporal Query Processing in Multiprocessor Database Machines," UCLA Computer Science Department, Technical Report CSD-910077, November 1991. Accepted for publication in the *18th International Conference on Very Large Data Bases.*

T.Y. Leung and R.R. Muntz, "Stream Processing: Temporal Query Processing and Optimization," UCLA Computer Science Department, Technical Report CSD-910079, December 1991. To appear in *Temporal Databases* as an invited book chapter, ed. R. Snodgrass.

Abstract of the Dissertation

Query Processing and Optimization in Temporal Database Systems

by

Ting Yu Leung
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1992
Professor Richard Muntz, Chair

There is a tremendous need for keeping evolving history of the "enterprise" of interest online, resulting in a *temporal* database. In this dissertation, we consider query processing and optimization aspects for temporal databases. Since storing temporal information in databases is a new application area, it is not surprising that the characteristics of temporal queries are much different from those of conventional queries. Conventional relational systems are often inefficient for temporal query processing because the new characteristics are not taken into consideration. As an example, a temporal join often contains a conjunction of several inequalities involving only time attributes. In conventional relational systems, this type of queries is processed using the nested-loop join algorithm, which may not be the most efficient method. However, it can be processed much more efficiently when new processing strategies are used.

We present a stream processing approach for temporal query processing. Given properly sorted data, implementation of temporal joins and semijoins as stream processors can be very efficient. We discuss the tradeoffs between sort orderings, the amount of local workspace, and multiple scans over input streams. Stream processing algorithms for various temporal joins and semijoins, and their workspace requirements for various data sort orderings are presented.

We propose a novel indexing technique for temporal data streams that is based on the stream processing techniques. The index can be exploited in processing complex multi-way joins that are qualified with snapshot operators (e.g., the "as of" operator). The advantages and limitations of the scheme and a quantitative analysis of the storage requirements are presented. We propose optimization alternatives that can reduce the storage requirements.

Multiprocessor database machines are probably more cost-effective at storing a huge volume of temporal data than centralized DBMSs. In this dissertation, we discuss issues involving temporal data fragmentation, temporal query processing, and query optimization in such an environment. We propose parallel strategies for multi-way joins which are based on partitioning relations on time attribute values, and optimizations for processing join queries qualified with comparison predicates involving time attributes. We analyze the schemes quantitatively, and show their advantages in computing complex temporal joins.

CHAPTER 1

Introduction

Many real-world database applications intrinsically involve time-varying information. Examples include inventory control in traditional business environments and many scientific applications which store measurement or trace data. With the availability of cheap processing and storage units, storing the evolving history of the "enterprise" of interest in a database system becomes a more attractive alternative than purging history data from the database.

These advanced applications, which often involve *temporal data analysis*, can be found in many areas such as in direct marketing. The analysis can provide valuable information which is crucial for decision making process. Conventional relational database systems, however, do not adequately support the storage and manipulation of a large volume of temporal data efficiently.

In the following sections, we first illustrate the importance of temporal DBMS support using several examples, which can be formulated as temporal pattern analysis problems. These examples suggest how a wide variety of similar applications can be supported by the temporal DBMS. We then contrast the conventional approach to supporting temporal functionality and our proposed approach.

1.1 Example Applications

1.1.1 Sales Revenues

In a traditional business environment, setting the future price of a product can very difficult. On the one hand, if the price is set too high, fewer items may be sold. On the other hand, if the price is too low, the profit may not be optimal. One strategy is to analyze the past sales performance in terms of sales volume and revenues over time, and using this information, attempt to do a better job in pricing the products. The following example taken from [Seg87] typifies the application domain. Suppose we have the relations:

Sales_volume(Item,Quantity,Date)
— sales volume of an item at a particular date
Price(Item,Amount,Start_date,End_Date)
— price of an item during the interval specified by the start and end dates.

Given the past sales records, one can easily obtain the revenues over time using a Quel-like query:

```
range of s is Sales_volume
range of p is Price
retrieve into Rev(Item=s.Item,Total=s.Quantity×p.Amount,Date=s.Date)
where s.Item=p.Item and p.Start_date≤s.Date and s.Date<p.End_Date
```

The point here is that storing history records can be very important in decision support systems, and the database systems should provide efficient accesses to history records and intelligent processing mechanism.

1.1.2 Convoy Detection

Monitoring the performance of computer systems or networks is often essential. It is desirable to support manual or automated intervention, if necessary, when certain phenomena are observed. Even if immediate intervention is not available, determining whether certain undesirable patterns occurred may still be valuable information.

Let us consider the convoy example from [Naq89]. Suppose a computer system is modeled as a network of queues. A node in the network represents some resources in the computer system. Entities of interest (such as requests of resources) arrive at nodes for services, and then they are routed to successor nodes for other services. Events of interests are collected in a database so that users can analyze the data by querying the database.

Suppose that we are interested in detecting if "convoys" exist in the system. Roughly speaking, a convoy can be viewed as a large number of entities (such as jobs) demanding resources at a node which "migrates" to other nodes over time. Because of a large number of requests moving from resource to resource,

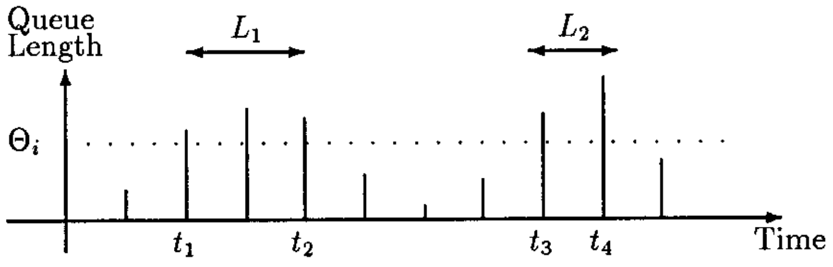


Figure 1.1: Observed lumps at node i

a convoy impacts the system performance considerably — existence of a convoy is an evidence of a type of correlation of job activities in the system which is an evidence of a "roving" imbalanced loading of resources. (Note that the long term averages may indicate a well balanced system with respect to the utilization of each resource and queue length.)

For simplicity, we describe convoys as the migration of lumpiness of entities from a node i to its successor node j. Lumpiness exists at node i during a period of time when the number of entities (i.e., its queue length) exceeds a threshold Θ_i during that period of time. That is, lumps are bounded by start and end times. Thus, in Figure 1.1 the pairs of times $[t_1,t_2)$ and $[t_3,t_4)$ are the time bounds for lumps L_1 and L_2 respectively. We also require that the interval length must exceed a threshold Δ_i before we call it a lump. The migration of a lump from a node i to node j is characterized by the overlap of their time bounds. Therefore, a convoy migration from a node i to node j is formulated as a pattern of two overlapping intervals. The convoy pattern is shown in Figure 1.2, where $[\mathrm{TS_i}, \mathrm{TE_i})$ and $[\mathrm{TS_j}, \mathrm{TE_j})$ are the time bounds for lumps in node i and j respectively.

When we detect a convoy, a number of actions can be taken. For example, we may further query the database about the types of entities in the convoy (e.g., what kinds of resources were requested most by the convoy of entities). This kind of information can be useful in improving the system performance in the future, e.g., we may want to re-route requests of a certain type to other nodes available for the same service, or even drop the requests from the system.

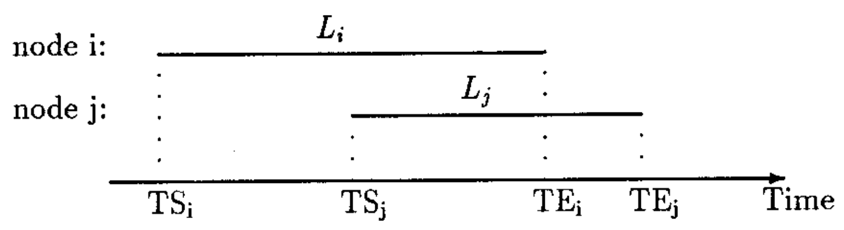


Figure 1.2: Characterization of a convoy migration from node i to its successor node j

1.2 This Research

A common approach to implementing a temporal DBMS is as follows:

> First, each tuple is augmented with a pair of time attributes which indicate its lifespan, and the temporal tuples are stored in a conventional relational DBMS. Second, a number of temporal operators are incorporated into the query language allowing users to query the time attributes (and thus the corresponding temporal data model is defined). A preprocessor which translates a temporal query (with temporal operators) into an equivalent relational query is implemented. The translated query is then processed by the relational DBMS which stores the temporal data.

That is, one can implement a temporal DBMS on the top of a relational database system. Most temporal operators are syntactic sugar — they can be directly specified in terms of comparison predicates and join predicates involving only time attributes; the use of these operators merely allows users to express a temporal query more intuitively. This leads to the following observation:

> In general, query optimizers do not search over all possible equivalent query plans for the minimal cost plan [Sel79]. Moreover, the query processing strategies that are implemented are based on what are expected to be the common types of queries and data characteristics. It is our belief that a major difference between temporal and conventional queries is in the types of queries that are common. Although we

can translate temporal queries into their equivalent relational counterparts, executing the translated queries on relational DBMS may be very inefficient because the translated queries contains constructs which are often ignored by conventional relational query processors and optimizers. Attention to the characteristics of temporal queries (as well as the temporal data) is therefore key to an efficient query processing algorithm and optimization.

In the following, we highlight the main points of our research direction. We first concentrate on the characterization of temporal queries and data. As we will demonstrate in this dissertation, there is no fundamental theoretical difference between a time attribute which stores relevant time information and an ordinary integer-based attribute such as employee numbers or social security numbers. However, temporal data and queries provide several unique characteristics for query processing. We will argue that ignoring these characteristics can result in orders of magnitude poorer performance.

We observe several characteristics of temporal queries and data. For example, a temporal query often involves patterns of events, and a temporal query often contains a conjunction of several inequalities over the time domain and no equality conditions. Such forms of queries are not common in non-temporal DBMS, i.e., given one of these queries, one can replace time attributes by some integer-based attributes and the "new" query will not contain a typical form of query qualification which has been ignored for optimization purpose in the past. For example, processing these queries in relational systems usually relies on the nested-loop join which is not always the best choice. These same query characteristics, however, enable us to devise efficient temporal join algorithms. In this dissertation, we propose new query processing methods for temporal databases and new optimization techniques. The new approaches take advantage of data sort orderings so that often relations are scanned only once for completing a temporal join operation.

Another common type of temporal query is the snapshot or interval query. These are temporal join queries that refer only to tuples in a small "time window" as opposed to the entire relation lifespan. To efficiently process this type of query, we propose an indexing technique such that tuples close to the "time window" are retrieved, i.e., we can avoid retrieving tuples that are "far away" from the query-specific interval.

Recently multiprocessor database machines are receiving more attention and we believe that taking advantage of parallel processing capability is also a key to effective use of temporal database systems. A strategy in multiprocessor database machines for processing an inequality temporal join of two relations is to dynamically and fully replicate the smaller operand relation, and again, this may probably be very expensive. We study various methods in partitioning and storing temporal data in such machines, and propose alternative processing algorithms and optimization strategies for temporal joins. We provide an analytical model for estimating the overhead of replicating tuples and show under which conditions our proposed strategies are preferable.

1.3 Organization of this Dissertation

This dissertation is organized as follows. In Chapter 2, we give an overview of related work in the area of temporal databases. In each individual chapter, we will present a more detailed comparison between our approach and the previous work, wherever it is appropriate.

Chapter 3 is devoted to a discussion of fundamental concepts and background information regarding temporal databases. We adopt a simple temporal data model whose characteristics and features can be found in many proposed temporal data models, and show that most temporal operators are equivalent to relational expressions.

In Chapter 4, we propose a stream processing approach to implementing temporal join and semijoin operators. The idea of stream processing techniques is to take advantage of data sort orderings such that (1) input relations are scanned only once and (2) the amount of local workspace (i.e., main memory) can be kept small.

In Chapter 5, we propose a generalized data stream indexing technique that can facilitate the processing of snapshot or interval queries. The idea is to provide an indexing mechanism such that tuples in proximity of the query-specific time interval or time point can be retrieved efficiently. A quantitative analysis of the storage requirement of the index and an optimization technique for reducing the storage size required are presented.

In Chapter 6, we focus on the processing and optimization of temporal join queries in multiprocessor database machines. Fragmentation schemes for temporal data are discussed and new parallel processing strategies are proposed. Optimization alternatives in reducing the tuples to be replicated across processors and implementation issues are presented.

Finally, we present our conclusions and directions of future research work in Chapter 7.

CHAPTER 2

Related Work

In this chapter, we present an overview of related work in the area of temporal databases. We then discuss related work in the area of temporal query processing and optimization.

2.1 Overview

Most research on temporal databases can be loosely categorized into four areas: semantics of time, logical data modeling, physical implementation, and deductive temporal databases. A rather complete bibliography can be found in [Soo91].

Semantics of Time The first area is the formulation of the semantics of time [All83, Cli83] and is closely related to research issues in knowledge representation[1]. [All83] describes, from the perspective of artificial intelligence, an interval based temporal logic and a reasoning algorithm based on constraint propagation. [Cli83] introduces a formal semantics for time in historical databases and a calculus based query language.

Logical Data Modeling The second area, which is the focus of many research activities, is the logical modeling of temporal data [Ben82, Cli85, Sno85, Sho86, Cli87, Seg87, Sno87, Gad88, Dut89, Tan89, Kaf90, Tuz90]. Many of these studies emphasize extending the relational data model to capture time semantics and to support relational temporal query languages; a notable exception is [Sho86, Seg87] which will be discussed in more details below. These extended models generally augment relations of the snapshot data model with several time

[1] There have been some debates on whether time interval or time point is more "natural" representation for temporal data, which is the basis for temporal data models [Lum84, Cli85, Sno85, Ahn86, Cli87].

attributes (such as ValidFrom and ValidTo attributes [Sno85]) which store the relevant time information[2]. New temporal operators are also defined in these extended data models (usually based upon traditional relational algebraic operators [Ull82]) to allow users to query time attributes but not update them directly. A detailed comparison of various relational algebras incorporating the time dimension can be found in [McK91].

[Sno85, Sno87] distinguish four categories of DBMS that support time attributes explicitly; the classification is based on the use of *transaction time* attributes and *valid time* attributes. The four categories are: *snapshot*, *rollback*, *historical*, and *temporal* DBMS. To illustrate the taxonomy, let us consider a simple example taken from [Sno85]. A snapshot relation is simply a table in a relational database system. For example:

Faculty(Name,Rank)

which stores tuples indicating the rank of a faculty member. This snapshot relation can be augmented with a pair of valid time attributes (ValidFrom and ValidTo) to represent the valid period during which a faculty member held a particular rank. The table now becomes:

Faculty(Name,Rank,ValidFrom,ValidTo)

which stores tuples indicating the rank of a faculty member during the period [ValidFrom,ValidTo) being modeled. The valid time attributes represent the time interval when the relationship or attribute being modeled was valid, and thus a database system that supports valid time is called a historical DBMS. Instead of augmenting the snapshot relation with valid time attributes, a pair of transaction time attribute (TransStart and TransEnd) can be added to represent the period during which a tuple was stored in the database. For example the table now becomes:

Faculty(Name,Rank,TransStart,TransEnd)

which stores tuples indicating that the rank of a faculty member was stored at time TransStart and was logically removed (but not physically removed) at time

[2] There have been different opinions on which "level" (such as attribute or tuple level) is more appropriate for augmenting the time attributes.

TransEnd. A database system that supports transaction time is called a rollback DBMS. A temporal DBMS [Sno85] supports both transaction time as well as valid time, i.e., it is both a rollback and historical DBMS. For example, the following is a temporal relation:

Faculty(Name,Rank,ValidFrom,ValidTo,TransStart,TransEnd).

In [Sno87], a language (TQuel) has been defined for data manipulation. Queries expressed in TQuel can be translated into Quel for processing.

The taxonomy in [Sno85] assumes that valid time attributes and transaction time attributes are completely unrelated. However, in some applications these time attributes do exhibit some interrelationships which can be used to further refine the taxonomy. In [Jen91] this assumption has been relaxed resulting in a framework for generalization and specialization of temporal relations. The authors list a number of possible relationships between the valid and transaction time attributes. For example, a temporal relation is called *retroactive* if the values of an item are valid before they are stored in the relation. There are several implications of this work. First, the framework allows a more precise characterization of temporal data, and thus allows us to enforce data integrity with respect to the interrelationships. Moreover, the more precise characterization may suggest alternative query processing approaches. For example, processing techniques for valid time attributes (such as the stream processing techniques that we propose in a later chapter) can be adapted to process queries involving both valid and transaction time attributes for certain specialized classes of relations.

In [Sho86, Seg87], a *time sequence* is proposed as the basic construct for logical modeling of temporal data. A time sequence is denoted as <S,(T,A)*> where S is the object surrogate, T is the time point, A is the attribute value, and (T,A)* is a sequence of time-value pairs. A time sequence is a time-ordered set of temporal values (i.e., time-value pair) for a single surrogate instance. In the model, the authors distinguish between the *time points* and the *data points* of a time sequence. The time points are all the potential points in time that can assume data values, while the data points are only the points that actually have data values. Several properties of time sequences, such as *time granularity*, *lifespan*, *regularity*, and *type* (e.g., stepwise constant, continuous and discrete) are identified. Below we give a brief summary of these properties.

Time granularity specifies the granularity of the time points of a time sequence. For example, two common representations are ordinal and calendar. In the ordinal representation, time points are simply considered as natural numbers (1, 2, 3, $\cdots$). In the calendar representation, the usual calendar time (such as year, month, day, etc.) is assumed.

Each time sequence is associated with a lifespan which specifies the range of valid time points of the time sequence. There are several approaches in specifying lifespans. For example:

- the range is specified by a start time and an end time, both of which are fixed.
- the range is specified by an end time (e.g., the current time) and a fixed distance to the past. In this case, one can imagine that the lifespan is a moving (fixed width) "window" along the time dimension as the current time is continuously advancing.
- the range is specified by only a fixed start time, i.e., the end time is continuously advancing.

The granularity of a time sequence is distinguished as follows: a regular time sequence contains a value for each time point during the entire lifespan, while an irregular time sequence contains data values for only a subset of time points.

We consider two common types of time sequences: discrete and stepwise-constant. For a discrete time sequence, as shown in Figure 2.1(a), each attribute value of an object is not related to other values and consequently missing data values cannot be interpolated. For instance, the number of copies of a book sold per day can be considered as a discrete time sequence. For stepwise-constant time sequences, as shown in Figure 2.1(b), the attribute value of an object at a time point t is the data value at the latest data point prior to t. That is, the data value between two consecutive data points can be computed using a stepwise-constant interpolation function. For example, the price of a book is a stepwise-constant time sequence. Note that the stepwise-constant function is often used implicitly as an extrapolation function for the current data value. For example, the price of a book remains the same since its last price change (i.e., the last data point is sometime in the past).

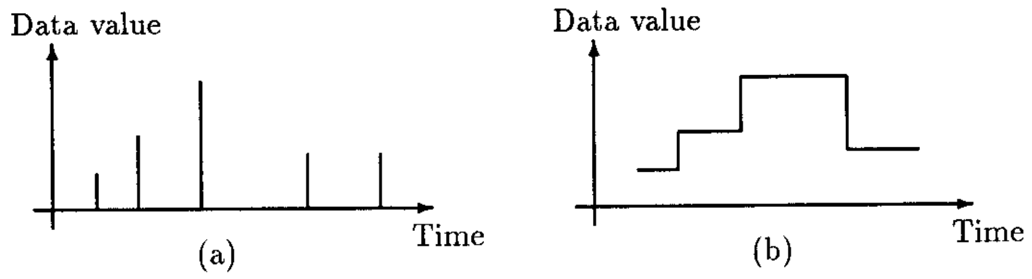


Figure 2.1: Time-varying attributes: (a) discrete time sequence and (b) step-wise-constant time sequence

In [Gad88] the author argues that time interval and time point based tuples are not appropriate for modeling time-varying attributes because both representations are not closed under union, intersection, and complementation operations. For example, the "complement" of a given time interval of a tuple with respect to the relation lifespan may produce two time intervals, and thus the time representation becomes a set of intervals instead of a single interval. It is proposed in [Gad88] that a *set* of time points (or equivalently a set of time intervals and/or time points for compact representation) for a data value is used. The rationale is that the time representation remains the same (i.e., a set of time points) upon the application of any relational algebra operator — the *closure* property. However, we note that this time representation does not increase the language expressiveness. Note that this time representation (set of time points) can be viewed as a specific time sequence in [Sho86, Seg87] where the data points with the same data value are "grouped" together as a tuple. The property of *homogeneity* is also discussed — if a relation contains several time-varying attributes, this property holds if the periods of validity of all the attributes in a given tuple are identical. A consequence is that one can represent such tuple more concisely using only a pair of time attributes which represents the period of validity.

Other proposed temporal data models include [Dut89] where the notion of probability of event occurrence (i.e., change of data values) is incorporated. Nested relations and complex objects are proposed for temporal data in [Tan89] and [Kaf90] respectively.

Physical Implementation The third area concerns physical implementation issues [Ben82, Lum84, Ahn86, Rot87, Gun89, Elm90, Kol91]; the focus is mainly on new access methods and data organization strategies in a centralized database environment. In the following, we briefly present several indexing techniques that are most recently proposed [Gun89, Elm90, Kol91].

To illustrate the concepts in Segment Index (SR-tree) [Kol91], we first consider the Segment Tree (ST-tree) in [Ben80] and the R-tree in [Gut84].

ST-tree is a main-memory based binary search tree whose major purpose is to provide means on efficiently finding all intervals (i.e., horizontal segments) that contain a given time point. An example ST-tree is shown in Figure 2.2. Given N horizontal line segments, the endpoint values are stored in sorted order in the leaf nodes. A non-leaf node represents an interval which contains the intervals represented by its left and right child nodes. A line segment is represented by "spanning" several nodes of the ST-tree. For example, the tuple e_1 whose interval is [2,5) spans the nodes with a dot (•) where e_1 is stored. Searching for all segments that contains a given point starts from the root of the ST-tree. The segments in the answer set are stored in the nodes along the path from the root to a leaf node that contains the given point. For example, as shown in Figure 2.2, segments e_3, e_1 and e_2 which contains the point 4 are stored along the path to the leaf node storing e_2.

The R-tree can be considered as an extension of B-trees for k-dimensional objects (e.g., 2-dimensional regions and horizontal line segments) [Gut84]. It is a balanced tree-structured index for representing objects using minimal bounding rectangles in k dimensions. Let us consider two-dimensional R-trees. Leaf nodes contain entries of the form:

(I,object-id)

where *object-id* is a pointer to a data object and I is a two-dimensional minimal rectangle which bounds the object. For rectangular objects, I will be the object itself. As shown in Figure 2.3, there are two leaf nodes: the left leaf node stores objects R_3, R_4, R_5, and R_6, while the right leaf node stores R_7, R_8, and R_9. Non-leaf nodes contain entries of the form:

(I,child-pointer)

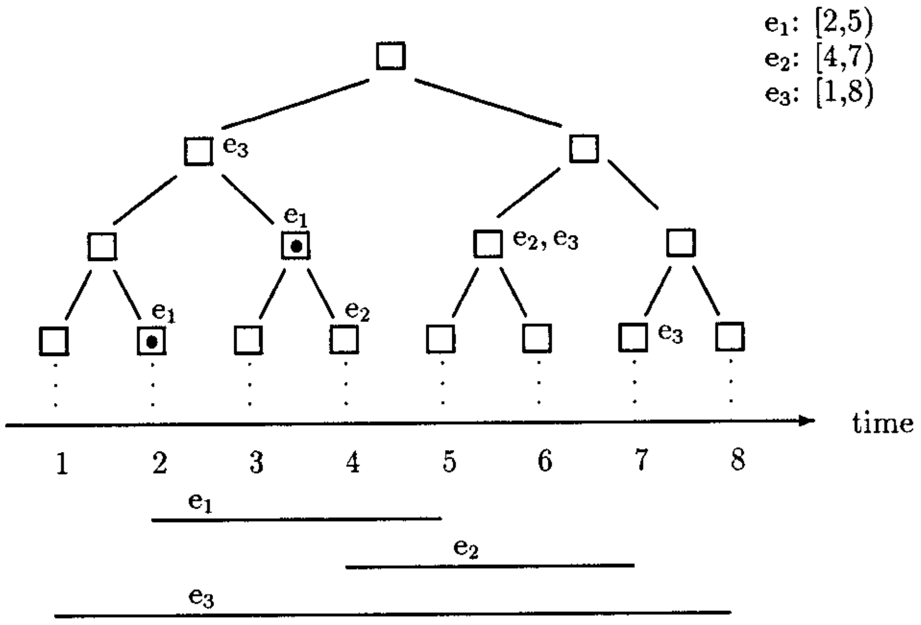


Figure 2.2: A segment tree

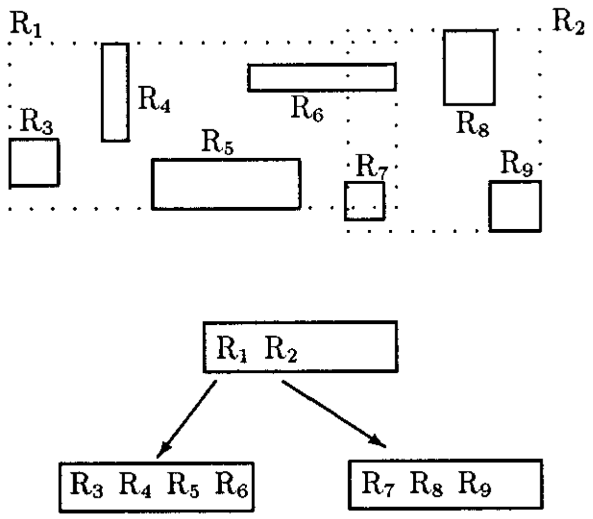


Figure 2.3: An R-tree

where *child-pointer* is a pointer to a child node in the next level and I is a minimal rectangle which bounds all the entries in the successor node. For example, the rectangle R_1 contains objects R_3, R_4, R_5, and R_6, while R_2 contains objects R_7, R_8, and R_9. Note that the pointer to a particular object is stored only once in the tree.

To find all objects that overlap with a given rectangle R, the search process starts from the root. If an entry in the node that is currently being searched overlaps with the rectangle R, the corresponding child node will also be searched. One of the important aspects of R-tree is the node-splitting algorithms. For example, if we insert an object which is bounded by R_1 in Figure 2.3, the left leaf node overflows and thus it will be split into two nodes. The goal of the node-splitting algorithm is to minimize the area covered by each individual new node as well as the overlapping area of the two new nodes. However, these two goals sometimes can be contradictory, meaning that if one minimize the area covered by each individual new node, the overlapping area may increase substantially, or vice versa. Some heuristic algorithms have been proposed for this node-splitting process [Gut84] which will not be covered here.

The SR-tree is a combination of the ST-tree and R-tree. The SR-tree extends the ST-tree strategy from binary trees in main memory to a multi-way disk-based file structure. The major features in the SR-tree is that segments which span lower level nodes may be stored in non-leaf nodes, and that node size may vary. The search algorithm follows a similar strategy to that for R-trees. For brevity, we omit the insertion (such as node splitting strategies) and deletion algorithms (see [Kol90, Kol91] for details).

In [Gun89] the Append-Only tree (AP-tree) is proposed. The tree structure which is a multiway search tree is a hybrid of an ISAM index and a B^+-tree. Figure 2.4, from [Gun89], is an example of an AP-tree of order 4. The leaf node contains the ValidFrom values in the relation and the associated pointers to data tuples. That is, the tree stores the left endpoints of all intervals in the relation. Searching the tree can be achieved via the root pointer or the rightmost leaf pointer[3]. The AP-tree differs from a B^+-tree on the ValidFrom attribute in the following way: all nodes in an AP-tree are packed with data values except in the rightmost subtree where data will be appended. That is, when a node overflows,

[3] It is implicitly assumed one can also access the leaf nodes via a leftmost leaf pointer.

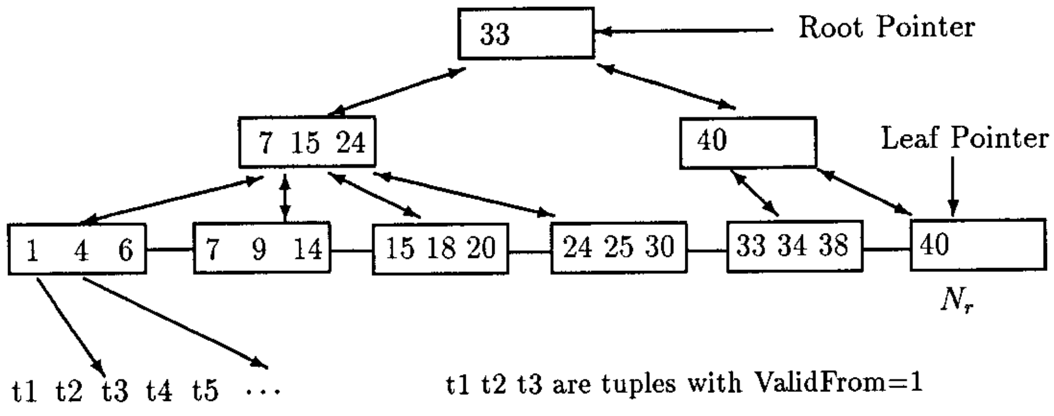


Figure 2.4: Append-Only tree of order 4

it will not be split into two nodes of half full (see the right most leaf node N_r containing the ValidFrom value 40 in Figure 2.4); a new node is appended to its right instead, and an index record is inserted into its parent node. In order words, the AP-tree is not a balanced tree. With the assumption that the database is "append-only" (i.e., data is static) and temporal tuples are inserted into the database in increasing order of the ValidFrom value, higher disk space utilization can be achieved by storing as many records as possible in a node until the node overflows.

The Time Index approach in [Elm90] stores all the ValidFrom and ValidTo attribute values (i.e., endpoint values of time intervals) in the relation in a B^+-tree. That is, the Time Index structure is a conventional balanced tree. The major characteristics is that at each endpoint value t, the tuple identifiers of all tuples that are active as of t are stored at the leaf nodes. An example of interval records and a time index is shown in Figure 2.5. For example, the lifespan of the tuple e_3 is $[t_2,t_5)$. At the endpoint t_5, tuples e_2, e_4 and e_6 are active and thus their tuple identifiers are stored in the leaf node accordingly. Note that the time index effectively stores the tuple identifiers of the entire snapshot of the relation as of an endpoint value, and therefore it requires significant storage space. A number of approaches have been proposed to reduce the storage space [Elm90, Elm91]. For example, the complete list of active tuples is kept for the first entry of each leaf node and only the incremental changes are kept for subsequent

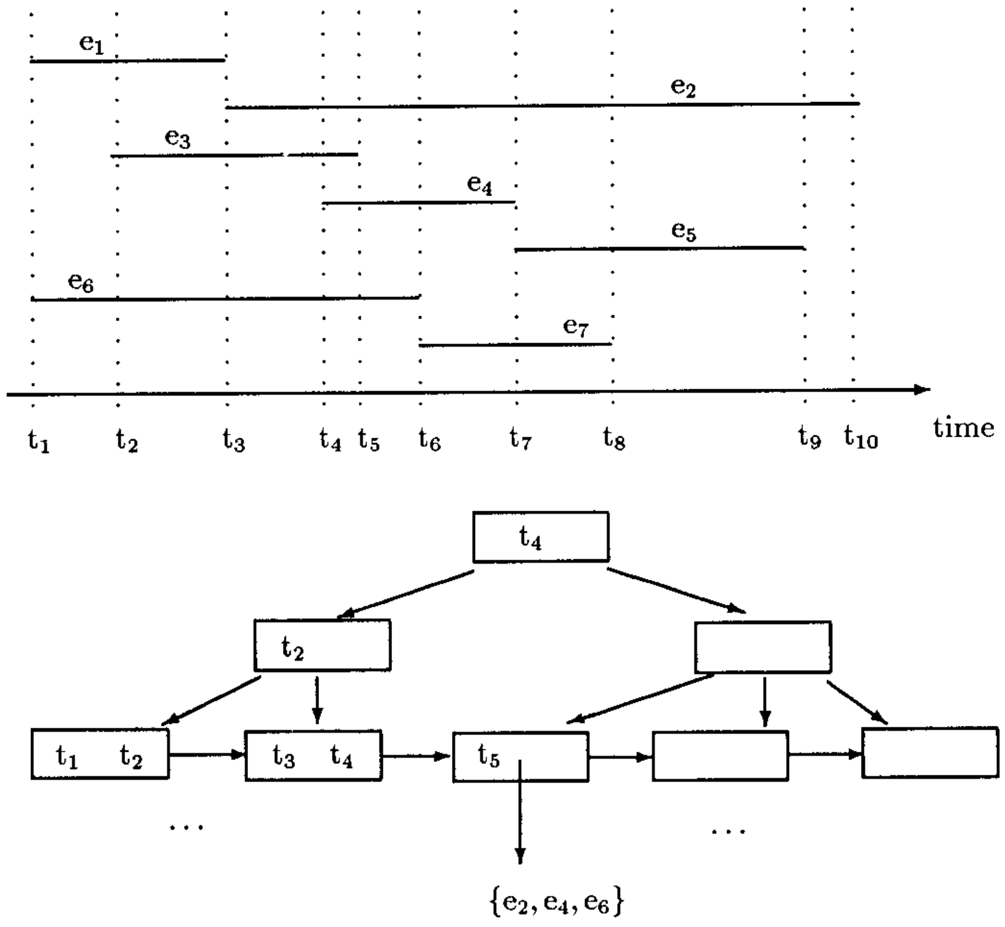


Figure 2.5: Sample interval records and a time index

entries in the same leaf node [Elm90] — the idea is to reduce duplication of tuple identifiers. Nonetheless, keeping a list of tuple identifiers of all active tuples at every endpoint value (i.e., every distinct ValidFrom and ValidTo values) is probably very expensive.

In [Kol90], a mixed-media indexing structure, which is designed in such a way that it spans mixed-media devices such as both magnetic and optical disks, has been proposed. The idea is that older history tuples can be migrated onto optical disks which are cheaper compared with magnetic disks while the pointers to those migrated tuples are kept on the magnetic disks. The file structure is, again, a variation of an R-tree [Gut84].

Deductive Temporal Databases The fourth area is the support of deductive capability in temporal databases [Cha88, Sri88, Cho90, Kab90]. [Cho90, Kab90] focus on the temporal aspects of deductive databases, and are more concerned with representing and querying periodic temporal data, i.e., possibly infinite data. [Cha88, Sri88] propose different approaches to represent and query temporal relationships (such as "before") among all objects or events. A graph model is proposed in [Cha88] to represent temporal relationships and to define temporal queries. The graph model enables the analysis of query processing complexity using the graph structure, and the generalized transitive closure is their main computation consideration. In [Cha88], only binary temporal relationships in the form of Temp(Event$_1$,Relationship,Event$_2$) are considered, e.g., <world_war_II,before,1950> is a tuple of this form. Based on event tuples of this form, a graph can be constructed; nodes in the graph are event identifiers (e.g., world_war_II) or absolute temporal values (e.g., 1950), and directed edges represent temporal relationships (e.g., "before"). Responses to queries such as finding all events that took place during the interval $[t_1, t_2)$ can be found using the constructed graph. Similarly, one can also find the temporal relationship between two events.

A different approach can be found in [Sri88] where the authors discuss the use of Event Calculus (i.e., first-order classical logic augmented with negation as failure) in formalizing the semantics of time in deductive databases, and in providing the capability of deducing temporal relationships as well as proactive/retroactive updates. Events may be either time interval based or time point based. For example, "possesses(mary,book)" represents the fact that mary possesses a book. For example, we denote e_1 and e_2 as an event in which "john gave a book to mary" and an event in which "mary gave the book to bob" respectively [Sri88]. These facts are represented by:

before(e_1, processes(john,book))
after(e_1, processes(mary,book))
before(e_2, processes(mary,book))
after(e_2, processes(bob,book))

Axioms (i.e., rules) can be stored and used for deduction capability. For example, one can deduce the existence of a time period during which "mary holds the book". Note that the axioms often depend on the domain of application.

Others Other research work related to temporal databases includes schema evolution [Ban87, Kim88, Nav88] and modeling dynamic aspects of information systems [Bar85, Lin87, Obe87, Ngu89]. Temporal logics such as [Lam83, Kow86] traditionally focus on complex temporal reasoning, but in [Lip87] a temporal logic is defined for specifying admissible dynamic behavior of database systems, i.e., how database state can evolve over time.

2.2 Processing Temporal Joins

In [Seg89], the event-join(X,Y) is defined as:

> TE-join(X,Y) ∪ TE-outerjoin(X,Y) ∪ TE-outerjoin(Y,X)

where the TE-outerjoin(X,Y) is defined as:

> For a given tuple $x \in$ X, outerjoin tuples (with null values) are generated for all time points t $\in$ [x.ValidFrom,x.ValidTo) where there does not exist $y \in$ Y such that both (i) t $\in$ [y.ValidFrom,y.ValidTo) and (ii) the join predicate "x.S=y.S" (i.e., involving only non-time attributes) are satisfied.

And the TE-join, known as the time-equijoin, is defined as:

> Two tuples from the joining relations qualify for concatenation if their time intervals intersect and the equality join predicate "x.S=y.S" hold.

The TE-join becomes a T-join (known as the time-join [Gun91]), when the equality join predicate is actually "true".

An event-join(X,Y) groups several temporal relations, each of which stores tuples of a time-varying attribute, into a single relation. A tuple in the resulting relation is created whenever at least one of those attributes are updated (a particular attribute may assume a null value). The uniqueness of this join is that it combines the time-equijoin and the outerjoin components into a single operation.

Consider an example borrowed from [Seg89] whose temporal relations are shown in Table 2.1. Assuming that the lifespan is [1,20), event-join(Manager, Commission) would produce a relation shown in Table 2.2.

Manager	E#	Mgr	TS	TE
	E1	Tom	1	6
	E1	Mark	9	13
	E1	Jay	13	20
	E2	Ron	1	19
	E3	Ron	1	20

Commission	E#	C_rate	TS	TE
	E1	10%	2	8
	E1	12%	8	20
	E2	8%	2	8
	E2	10%	8	20

TS ≡ ValidFrom

TE ≡ ValidTo

Table 2.1: Employees' managers and their commission rates

Mgr_Com	E#	Mgr	C_rate	ValidFrom	ValidTo
	E1	Tom	*null*	1	2
	E1	Tom	10%	2	6
	E1	*null*	10%	6	8
	E1	*null*	12%	8	9
	E1	Mark	12%	9	13
	E1	Jay	12%	13	20
	E2	Ron	*null*	1	2
	E2	Ron	8%	2	8
	E2	Ron	10%	8	19
	E2	*null*	10%	19	20
	E3	Ron	*null*	1	20

Table 2.2: Result of event-join(Manager,Commission) in Table 2.1

The authors note that the most difficult step in processing the event-join is to determine the interval during which some attributes assume null values since it is assumed that null values are not stored in the temporal database. For example, in order to produce the first tuple <E1,Tom,*null*,1,2> in Table 2.2, one would have to determine that E1 has a data value (i.e., "Tom") stored in the relation Manager but E1 does not have a data value in the relation Commission during the interval [1,2) (see Table 2.1). In [Seg89], several algorithms are proposed for processing event-join. Here we briefly outline the modified sort-merge algorithm which appears to perform better and is closely related to our approaches; readers may refer to [Seg89] for details of the modified nested-loop join as well as the join algorithms using the Append-Only Tree structure described earlier.

For the event-join(X,Y) where X and Y stand for X(S,U,ValidFrom,ValidTo) and Y(S,V, ValidFrom,ValidTo) respectively, the approach is to sort both relations by S as the primary order and then by ValidFrom as the secondary order. If there exists an object s which has a tuple in X (respectively Y) but not in Y (respectively X), the event-join generates an outerjoin tuple with a null value for s. When the object s has some tuples in both X and Y, the join algorithm determines the time interval during which an attribute assume a null value. As tuples of the same object are sorted by ValidFrom values, one can easily determine the intersection interval as well as the time interval for the outerjoin tuple (if any). For example, suppose the first tuple of an object s in both relations are denoted as x_s and y_s, and "x_s.ValidFrom<y_s.ValidFrom and y_s.ValidFrom<x_s.ValidTo" holds, then an outerjoin tuple whose interval is [x_s.ValidFrom,y_s.ValidFrom) is generated because no subsequent Y tuple (that have not been read) will have a smaller ValidFrom value than x_s.ValidFrom. The join process completes when all tuples in both relations have been read.

There are several comments. First, the join algorithm for TE-join when both relations are sorted by the surrogate (as the primary sort order) and then by ValidFrom (as the secondary sort order) is similar to the above event-join algorithm except that the outerjoin tuples are not generated. Second, in the event-join and TE-join algorithms, both relations are scanned only once and only one tuple from each relation is kept in the buffer for comparison. Lastly, the authors consider null values in the event-join operation but the role of null values in a select operation is not defined accordingly. In other words, in order to process a query involving both the event-join as defined above and some select predicates, the event-join algorithms must be used prior to applying select operations.

2.3 Definitions

We now define several terms that are used frequently in this dissertation.

Definition 2.1 "A op c", where A is an attribute, op is a relational operator ($>$, $\geq$, $=$, $<$, $\leq$, $\neq$) and c is a constant, is called a *comparison predicate.* Similarly, "A op B" for attributes A and B is called a *join predicate.* A subclass of join predicates is called *temporal join predicate* if both attributes A and B are time attributes. □

Definition 2.2 A *query qualification* in conjunctive normal form is a conjunction ($\wedge$) of several join predicates and comparison predicates. A query qualification involving relations $R_1, \cdots, R_m$, $m \geq 1$, is denoted as $P(R_1, \cdots, R_m)$ or simply P. □

Definition 2.3 Given numbers $a_1, \cdots, a_m$, for some $m>1$, $\mathsf{max}(a_1, \cdots, a_m)$ returns a_j such that $a_j \geq a_i$, where $1 \leq j \leq m$ and $1 \leq i \leq m$ while $\mathsf{min}(a_1, \cdots, a_m)$ returns a_j such that $a_j \leq a_i$ where $1 \leq j \leq m$ and $1 \leq i \leq m$. □

CHAPTER 3

Temporal DBMS vs Relational DBMS

In this chapter, we discuss the fundamental concepts and background information regarding temporal databases. Particularly, we address a very important issue, namely, the differences between a temporal DBMS and a relational DBMS.

This chapter is organized as follows. In Section 3.1, we present the temporal data model that we adopt in this dissertation, the "append-only" type of updates to temporal data and the statistical information. Although we note that there is no theoretical difference between time attributes and integer-based attributes, we show in Section 3.2 that there are several interesting characteristics of temporal data and queries that we should consider for more efficient temporal query processing and optimization. In Section 3.3, we show that temporal operators proposed in many literatures can actually be expressed in relational expressions or relational queries. This verifies our conjecture that temporal operators are really syntactic sugar for query qualification, and thus do not increase the power of the query language. In other words, most (if not all) temporal data models proposed in the past are equivalent to relational data model. Finally, we discuss the classes of queries that we consider in this dissertation in Section 3.4.

3.1 Background Information

3.1.1 Temporal Data Model

We adopt a modified version of the *time sequence* concept in [Sho86, Seg87] as the basic data construct in our temporal data model[1]. We treat time as a sequence of discrete, consecutive, equally-distanced points $\{t_o, t_1, \cdots, now\}$, where *now*

[1] Other temporal data models such as [Cli85, Sno87] which employ one or more time attributes to represent time-varying information would suffice. However, temporal query processing and optimization are largely ignored in their work, which is the main focus of this dissertation.

is a special marker that represents the current time. That is, time points are totally ordered and are monotonically increasing. The sequence of time points can simply be treated as isomorphic to the natural numbers, and therefore we do not specify the time unit.

For stepwise-constant time-varying attributes, we use a *time-interval* tuple, <S,V,TS,TE>, where S is the identity of the object, V is a continuous time-varying attribute of concern, TS and TE represent *valid start time* and *valid end time* respectively. That is, TS and TE are ValidFrom and ValidTo time attributes respectively. Semantically, a tuple <S,V,TS,TE> represents an object S with the attribute value V during the period [TS,TE). Naturally, within a tuple the TS value is always smaller than the TE value. For discrete time-varying attributes, we use triplets <S,V,TA>, called *time-point* tuples, where TA is a time attribute.

We assume that all temporal relations have a homogeneous lifespan — [0,*now*). Furthermore, it is assumed that a time-varying attribute can have at most one data value at any point in time. Note that the time-varying attribute V in both stepwise-constant and discrete time sequence can be generalized as a list of attributes as opposed to a single attribute.

In the data model, the TS, TE and TA are referred to as time attributes (or simply *timestamps*) while other attributes are referred to as non-time attributes. A time-interval temporal relation is a set of time-interval tuples while a time-point temporal relation is a set of time-point tuples. In this dissertation, we will focus mainly on time-interval temporal relations. Unless ambiguity arises, we refer time-interval temporal relations to as temporal relations.

Using the taxonomy in [Sno85], the TS and TE timestamps are called the *effective* or *valid* timestamps as opposed to the *transaction* timestamps, and a database system which handles valid times is called a "historical database". It is our view that supporting transaction time and valid time share many common features. Particularly, our query processing algorithms and optimization strategies can be equally applicable in both rollback DBMS and historical DBMS described in the previous chapter. The readers should bear in mind that we are dealing with a "historical database" although we use the term "temporal database" as temporal data refers to both current data and history data. In this dissertation, we use the terms "temporal databases" and "historical databases" interchangeably.

	⋮		
smith	assistant	ts_1	te_1
smith	associate	ts_2	te_2
smith	full	ts_3	te_3
	⋮		

Table 3.1: A sample Faculty relation

An example of temporal relations is Faculty(Name,Rank,TS,TE)[2]. Together with the following integrity constraints and assumptions, this example is used in a subsequent chapter for illustration purposes. Name is the identity of a faculty member. For attribute Rank, we consider only three different ranks — assistant, associate and full. We assume in this example that an assistant professor can be promoted only to an associate professor and then to a full professor. In other words, there is a chronological ordering among the data values that the Rank attribute can assume. For the same faculty member, e.g., "smith" as illustrated in Table 3.1, "$te_1 \leq ts_2$" and "$te_2 \leq ts_3$" must hold. The interval [TS,TE) of a tuple is the time during which the faculty member holds the indicated rank. We also assume that a faculty member is at exactly one rank at any time between becoming an assistant professor and termination as a full professor. As we mention above, for any tuple x, "x.TS<x.TE" always holds.

3.1.2 Updating Temporal Data

In the data model, we consider the "append-only" update policy:

1. Time-point relations — when the attribute U of an object s has a new data value u at time t_a, a tuple $<s,u,t_a>$ is inserted.

2. Time-interval relations —

[2] The example is borrowed from [Sno87]. Also, the relation is in the first temporal normal form (1TNF) [Seg88].

(a) When a data value v (of attribute V) of an object s is valid at time t_s, a tuple $<s,v,t_s,now>$ is inserted into the relation.

(b) When the data value v is no longer valid at time t_e, the TE timestamp of the tuple (i.e., "*now*") is updated to t_e. i.e., the tuple becomes $<s,v,t_s,t_e>$[3].

That is, we only consider insertion of tuples and modification of *now* in the model.

We can now define *current* and *history* tuples of time-interval temporal relations, and data streams as follows.

Definition 3.1 A *current tuple* is defined as a tuple whose TS value is a specific time point and whose TE value is "*now*". A ***history tuple*** is defined as a tuple whose TS and TE values are specific time points (i.e., its TE value is not "*now*"). □

Definition 3.2 A *data stream* is defined as an ordered sequence of data objects or tuples. □

Unless ambiguity arises, data streams usually refer to time-interval temporal relations sorted on either TS or TE timestamp in this dissertation.

Two situations naturally occur in which tuples can be organized as a data stream in the append-only databases:

1. Current and history tuples are stored in the same file structure: whenever a tuple (i.e., $<s,v,t_s,now>$) is created, the tuple is appended to the data stream. When the data value (v) is no longer valid, say at time point t_e, the TE value of the existing tuple in the data stream is then modified to t_e. In this approach tuples are sorted by the TS values in increasing order.

2. Current and history tuples are stored in the different file structures: whenever a tuple (i.e., $<s,v,t_s,now>$) is created, the tuple is inserted into a table

[3] Modifying an existing data value corresponds to updating the TE timestamp of the existing tuple followed by inserting a new tuple for the new data value.

that stores only current tuples, i.e., in a current store. When the data value (v) is no longer valid at time point t_e, the TE value of the tuple is modified to t_e: the history tuple ($<s,v,t_s,t_e>$) is then removed from the current store and appended to the data stream. In this approach, tuples in the data stream are sorted by the TE values in increasing order.

Tuples in a data stream can be stored using a variety of file structures such as sequential file and B^+tree although different file structures generally have different retrieval and storage cost. The most important requirement is that tuples in a data stream can be efficiently accessed one at a time and in the order of successive timestamp values.

3.1.3 Temporal Data Statistics

Temporal data has several characteristics and statistical properties that are of interest here. These characteristics and statistics represent valuable information that provides some guidelines on evaluating a proposed scheme.

For time-interval temporal relations, time-varying attributes (or similarly relationships) can also be classified as *continuous* or *non-continuous* [Seg87]. A continuous time-varying attribute must have a valid non-null value during the object's lifespan, whereas a non-continuous attribute may not have a valid value (or equivalently it has a null value) during some period of time. For example, the stock price can be considered as a continuous time-varying attribute (types A and B in Figure 3.1) while a person checking out a book from a library can be a non-continuous time-varying relationship (type C in Figure 3.1).

In addition to the above characteristics, we will make use of several statistics. First, the relation lifespan (denoted as TR_{ls}) is assumed to be $[0,now)$, and is continuously expanding. The effective lifespan of an individual tuple, which is mentioned earlier, is specified by the pair of timestamps — [TS,TE). The average tuple lifespan is represented by $\overline{T_{ls}}$ and the average time between two consecutive insertions of tuples to a relation is represented by $\overline{T_{ins}}$, as depicted in Figure 3.1. The mean rate of insertion of tuples into a relation (denoted as λ) is $1/\overline{T_{ins}}$. For continuous attributes, these two figures are directly related: $\overline{T_{ins}}$ is $\overline{T_{ls}}/\overline{Ns}$, where $\overline{Ns}$ is the average number of surrogates in the temporal relation. These statistics will be used to characterize the performance of the proposed schemes

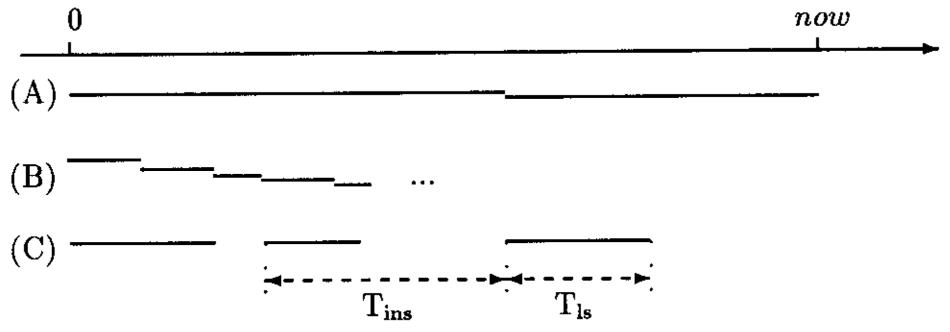


Figure 3.1: Characterization of temporal data

to be discussed later.

3.2 Timestamps vs Ordinary Attributes

From a theoretical view point there is no fundamental difference between timestamps and integer valued attributes such as the quantity of a product ordered and department numbers. However there are significant practical distinctions with respect to the manner in which temporal data is updated and queried. In the following, we list their major distinctions. We would like to emphasize that some of the distinctions may have been pointed out by other researchers, and the list is not necessarily complete.

1. Time is advancing in one direction.

   The time domain is continuously expanding. One can imagine that the value of the most recent time point (*now*) is monotonically increasing.

2. The constraint "R.TS<R.TE" holds for every time-interval temporal tuple.

   Naturally it is assumed that for each tuple its TS value must be smaller than its TE value. While most researchers implicitly make this assumption, it is seldom pointed out that this assumption can play a role in query processing and optimization (see Chapter 4).

3. Types of queries may be different.

   Temporal queries share many common operators with conventional queries such as select and natural join. The following highlights the major differences which are more in the nature of characterizing the types of temporal queries that might be expected and which would be more rare for non-temporal database systems. As we show in a later section, these queries can be expressed in terms of traditional relational algebra or query languages such as SQL and QUEL, and are often ignored by conventional query processors in terms of optimization.

- The join condition often contains a number of *inequality* join predicates involving only timestamps.
- A special kind of select query, commonly called *snapshot* or *interval* query, allows users to "view" the database content that is active over a period of time or as of a particular time.
- "within" operator — This operator represents the "distance" relationship between two entities. For example, find all events that occur within 5 minutes from the time an event X occurred.

The workload characteristics generally have a significant impact on the data organization. For example, for applications in which temporal data is more frequently accessed via surrogate values, tuple retrieval via surrogates should be as efficient as possible, e.g., "chaining" tuples of the same surrogates together as suggested in [Ahn86].

4. Meta-data (i.e., statistics, properties, and characteristics) of temporal data.

   The most commonly mentioned meta-data includes lifespan, time granularity, and regularity of temporal data along the time dimension. The meta-data of a relation can be significantly altered after an operator is applied to the relation. For example, it is pointed out in [Cli87, Seg87] that the lifespan may be changed as a result of temporal qualification. In this dissertation, meta-data is not our major concern; readers may refer to [Cli87, Seg87] for more details.

5. Temporal data update characteristics.

   Temporal data can be classified as "static" or "dynamic". "Static" means that once a piece of data is inserted into the database, it will not be updated. Otherwise, it is "dynamic". In general, history data is usually static in nature although some literature suggests supporting retroactive updates (e.g., [Lum84]). Proactive update is another proposed feature that is seldom found in conventional database systems. In most work, the append-only update policy is adopted. Moreover, users cannot update the timestamps arbitrarily but users can query timestamp values. Coupled with the fact that time is advancing in one direction, this kind of update suggests that a special storage structure that exploits the append-only policy may be more efficient (e.g., [Ahn88, Gun89])[4].

6. The special markers "*now*" are stored in current tuples.

   In general, the effective update times are not necessarily monotonically increasing with respect to the order of updates (e.g., due to concurrency control systems). The marker "*now*" of a tuple can be set to an arbitrary value although it is generally assumed that "*now*" is the latest current time. More importantly, the marker "*now*" cannot be treated as if it were the largest value in the time domain which is continuously expanding[5]. For this reason, the comparison between "*now*" and any other data values have to be defined accordingly. For example, given a time point t, one has to define what the comparison predicate "R.TE$<t$" means for current tuples (i.e., those whose TE value is "*now*").

7. Temporal data can be partitioned into the current and history versions.

   There is a natural separation of temporal data into current and history data. Current tuples tend to be more frequently accessed than history tuples, especially in business applications. Moreover, due to the append-only update policy, the current tuples are always modified when time-varying attribute values are changed. This distinction may suggest using a different storage structure (and storage media) for history and current tuples (e.g., [Ahn88]). For example, storing current tuples using a separate file structure allows us to eliminate storing the special markers "*now*" and therefore conventional indexing techniques can be used for current data (e.g., indexing the TS timestamp or other non-time attributes). Note that the TS and

[4] This also suggests that if retroactive update is not supported, one can store as many tuples in a disk page as possible so that higher disk utilization can be achieved. Generally, indices using dynamic splitting algorithms tend to lower the disk utilization.

[5] The marker "*now*" can be viewed as an unbounded variable in a logic programming language such as Prolog [Ste86]. Once it is set to a value, it cannot be changed.

TE values for all history tuples are known since "*now*" is not stored in any history tuple.

8. Time-varying attribute values can be continuously varying.

   The data values of some time-varying attributes can be represented by a function of time. For example, consider the position of a moving vehicle. Suppose that instead of storing a data value for every time point, we store the initial position and the speed of each vehicle. The current position of a vehicle can be expressed as:

   $$\text{current\_position} = \text{initial\_position} + \text{speed} \times \text{time\_elapsed}.$$

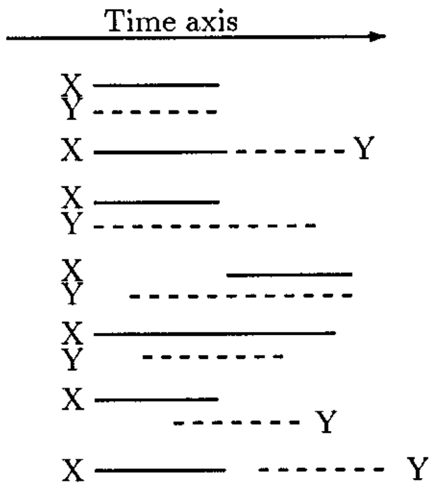
   That is, the current position of a vehicle can be computed using the extrapolation function. This distinction is seldom discussed or even addressed in temporal database research work, but it appears in the area of simulation and temporal deductive databases [Kab90, Nar89][6].

To summarize, although there is no theoretical distinction between a timestamp and an ordinary integer-based attribute (i.e., there is no fundamental difference between temporal DBMS and relational DBMS), making use of these characteristics of temporal data and queries, as we will argue, are essential to the efficient implementation of temporal DBMS.

3.3 Temporal Operators

In this section, we discuss several temporal operators that are commonly used in temporal DBMS literature; they are temporal join, "within", select, "time-project", and "time-union" operators. We show that except for the "time-union" operator, which returns a single interval that is equivalent to several overlapping or contiguous intervals, these operators can be expressed in terms of relational algebra or relational query languages such as SQL and QUEL.

[6] This type of extrapolation function can also be found in spatial databases. Moreover, one can think of temporal relation of this form contains (theoretically) infinite number of tuples.



Temporal Operators	Explicit Constraints
(1) X equal Y	X.TS=Y.TS ∧ X.TE=Y.TE
(2) X meet Y	X.TE=Y.TS
(3) X start Y	X.TS=Y.TS ∧ X.TE<Y.TE
(4) X finish Y	Y.TS<X.TS ∧ X.TE=Y.TE
(5) X contain Y	X.TS<Y.TS ∧ Y.TE<X.TE
(6) X overlap Y	X.TS<Y.TS ∧ Y.TS<X.TE ∧ X.TE<Y.TE
(7) X before Y	X.TE<Y.TS

TS ≡ ValidFrom, TE ≡ ValidTo
Integrity Constraints: X.TS<X.TE ∧ Y.TS<Y.TE

Figure 3.2: The 13 possible time-interval temporal relationships

3.3.1 Join Operators

Allen [All83] presents thirteen elementary temporal operators of time-intervals which are listed in Figure 3.2. We note that these temporal join operators, which often contain inequality predicates involving only timestamps, are just syntactic sugar for the query-specific constraints that are given in the right hand column of Figure 3.2. Note that the overlap operator is asymmetric with respect to the operands, and one can define a symmetric version as follows:

intersect-join(X,Y) — "X.TS<Y.TE ∧ Y.TS<X.TE".

In [Cli87, Gun91], the time-join (denoted as T-join) and the time-equijoin which is also called the natural time-join (denoted as TE-join) have been proposed. In [Gun91] the TE-join is defined as:

> Two tuples from the joining relations qualify for concatenation if their time intervals intersect and the equality join predicate P ("x.S=y.S") on only non-time attributes hold.

The TE-join is a T-join when the equality join predicate P is actually "true". In [Gun91], it is noted that "the concatenation of tuples is non-standard, since only one pair of TS and TE attributes is part of the two joining tuples". It turns out that both T-join and TE-join can actually be expressed in terms of the standard relational operators as follows:

$$
\begin{aligned}
& \pi_{\text{L,Y.TS,X.TE}} \left(\sigma_{\text{P} \wedge \text{X.TS} \leq \text{Y.TS} \wedge \text{Y.TS} < \text{X.TE} \wedge \text{X.TE} < \text{Y.TE}}(\text{X,Y}) \right) \\
\cup\ & \pi_{\text{L,Y.TS,Y.TE}} \left(\sigma_{\text{P} \wedge \text{X.TS} \leq \text{Y.TS} \wedge \text{Y.TE} \leq \text{X.TE}}(\text{X,Y}) \right) \\
\cup\ & \pi_{\text{L,X.TS,X.TE}} \left(\sigma_{\text{P} \wedge \text{Y.TS} < \text{X.TS} \wedge \text{X.TE} \leq \text{Y.TE}}(\text{X,Y}) \right) \\
\cup\ & \pi_{\text{L,X.TS,Y.TE}} \left(\sigma_{\text{P} \wedge \text{Y.TS} < \text{X.TS} \wedge \text{X.TS} < \text{Y.TE} \wedge \text{Y.TE} < \text{X.TE}}(\text{X,Y}) \right)
\end{aligned}
$$

where X(S,U,TS,TE) and Y(S,V,TS,TE) are temporal relations, L is a projection list (X.S, X.U, and Y.V), and P is a join predicate "X.S=Y.S". Suppose we are interested in only the tuple pairs that satisfy the join condition, then the TE-join and T-join become the intersect-join as opposed to the union of four joins:

$$\sigma_{\text{P} \wedge \text{intersect-join(X,Y)}} (\text{X,Y}).$$

That is, the query response consists of tuple pairs whose participating tuples intersect and satisfy P.

In [Seg89], the event-join(X,Y) is defined as:

TE-join(X,Y) ∪ TE-outerjoin(X,Y) ∪ TE-outerjoin(Y,X)

where the TE-outerjoin(X,Y) is defined as[7]:

> For a given tuple $x \in$ X, outerjoin tuples (with null values) are generated for all time points t $\in$ [x.TS,x.TE) where there does not exist $y \in$ Y such that t $\in$ [y.TS,y.TE) and the join predicate "x.S=y.S" (i.e., involving only non-time attributes is satisfied.

As in the case for the TE-join, the TE-outerjoin can also be defined in terms of relational algebraic operators. Below we show that the TE-outerjoin can be defined in traditional tuple calculus. From the relation X(S,U,TS,TE), we first obtain a new relation (denoted as $\overline{\text{X}}$) which contains tuples of null values (of

[7] Note that the TE-outerjoin is really not the same as the "outerjoin" operator defined in [Cod79].

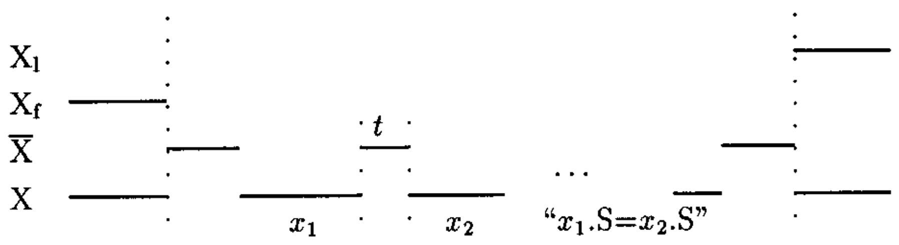


Figure 3.3: Deriving the relations $\overline{X}$, X_f and X_l from X

attribute V) for each surrogate in X that is not explicitly stored in X, as illustrated in Figure 3.3 [8]:

$$\begin{aligned}\overline{X} \equiv \{\ & t\text{<S,U,TS,TE>} \mid \exists x_1\ \exists x_2\ (\ x_1 \in X \wedge x_2 \in X \wedge x_1.\text{TE}<x_2.\text{TS} \wedge \\ & x_1.\text{S}=x_2.\text{S} \wedge t.\text{S}=x_1.\text{S} \wedge t.\text{V}=\textit{null} \wedge t.\text{TS}=x_1.\text{TE} \wedge t.\text{TE}=x_2.\text{TS} \\ & \wedge \neg\ \exists x_3\ (\ x_3 \in X \wedge x_1.\text{S}=x_3.\text{S} \wedge \\ & \qquad [x_3.\text{TS},x_3.\text{TE})\ \textsf{intersect}\ [\text{t.TS,t.TE})\ )\ )\ \}.\end{aligned}$$

Next, we obtain from X another relation (denoted as X_f) which contains the "first" tuple (of attribute V) for each surrogate in X:

$$\begin{aligned}X_f \equiv \{\ t\text{<S,U,TS,TE>} \mid \exists x_1\ (\ & x_1 \in X \\ & \wedge\ t.\text{S}=x_1.\text{S} \wedge t.\text{V}=x_1.\text{V} \wedge t.\text{TS}=x_1.\text{TS} \wedge t.\text{TE}=x_1.\text{TE} \\ & \wedge \neg\ \exists x_2\ (\ x_2 \in X \wedge x_1.\text{S}=x_2.\text{S} \wedge x_2.\text{TE}\leq t.\text{TS}\ )\ )\ \}.\end{aligned}$$

Lastly, we obtain from X another relation (denoted as X_l) which contains the "last" tuple of attribute V for each surrogate in X:

$$\begin{aligned}X_l \equiv \{\ t\text{<S,U,TS,TE>} \mid \exists x_1\ (\ & x_1 \in X \\ & \wedge\ t.\text{S}=x_1.\text{S} \wedge t.\text{V}=x_1.\text{V} \wedge t.\text{TS}=x_1.\text{TS} \wedge t.\text{TE}=x_1.\text{TE} \\ & \wedge \neg\ \exists x_2\ (\ x_2 \in X \wedge x_1.\text{S}=x_2.\text{S} \wedge t.\text{TE}\leq x_2.\text{TS}\ )\ )\ \}.\end{aligned}$$

Similarly we can obtain three new relations ($\overline{Y}$, Y_f, and Y_l) from relation Y. Using these six new temporal relations, the TE-outerjoin(X,Y) and the TE-outjoin(Y,X) are:

[8] We assume that a time-varying attribute can have at most one data value at any point in time.

TE-outerjoin(X,Y)
$\equiv$ TE-join(X,$\overline{\text{Y}}$) $\cup$ TE$_f$-join(X$_f$,Y$_f$) $\cup$ TE$_l$-join(X$_l$,Y$_l$) $\cup$ TE$_o$-join(X,Y)

TE-outerjoin(Y,X)
$\equiv$ TE-join(Y,$\overline{\text{X}}$) $\cup$ TE$_f$-join(Y$_f$,X$_f$) $\cup$ TE$_l$-join(Y$_l$,X$_l$) $\cup$ TE$_o$-join(Y,X)

That is, the TE-outerjoin is the union of four joins. The first three joins account for the cases in which a surrogate appears in both relations X and Y, while the last join (i.e., TE$_o$-join) accounts for the case in which a surrogate appears only in one relation (but not in the other relation). The joins TE$_f$-join(X,Y), TE$_l$-join(X,Y), and TE$_o$-join(X,Y) are defined as follows:

$$
\begin{aligned}
\text{TE}_f\text{-join}(\text{X}_f,\text{Y}_f) \equiv \{ & t\text{<S,U,V,TS,TE>} \mid \exists x\ \exists y\ (\ x \in \text{X}_f \wedge y \in \text{Y}_f \\
& \wedge x.\text{S}=y.\text{S} \wedge x.\text{TS}<y.\text{TS} \wedge y.\text{TS}\leq x.\text{TE} \\
& \wedge t.\text{S}=x.\text{S} \wedge t.\text{V}=x.\text{V} \wedge t.\text{U}=null \wedge t.\text{TS}=x.\text{TS} \wedge t.\text{TE}=y.\text{TS}\ )\ \} \\
& \cup \\
& \{\ t\text{<S,U,V,TS,TE>} \mid \exists x\ \exists y\ (\ x \in \text{X}_f \wedge y \in \text{Y}_f \\
& \wedge x.\text{S}=y.\text{S} \wedge x.\text{TE}<y.\text{TS} \\
& \wedge t.\text{S}=x.\text{S} \wedge t.\text{V}=x.\text{V} \wedge t.\text{U}=null \wedge t.\text{TS}=x.\text{TS} \wedge t.\text{TE}=x.\text{TE}\ )\ \}
\end{aligned}
$$

$$
\begin{aligned}
\text{TE}_l\text{-join}(\text{X}_l,\text{Y}_l) \equiv \{ & t\text{<S,U,V,TS,TE>} \mid \exists x\ \exists y\ (\ x \in \text{X}_l \wedge y \in \text{Y}_l \\
& \wedge x.\text{S}=y.\text{S} \wedge x.\text{TS}\leq y.\text{TE} \wedge y.\text{TE}<x.\text{TE} \\
& \wedge t.\text{S}=x.\text{S} \wedge t.\text{V}=x.\text{V} \wedge t.\text{U}=null \wedge t.\text{TS}=y.\text{TE} \wedge t.\text{TE}=x.\text{TE}\ )\ \} \\
& \cup \\
& \{\ t\text{<S,U,V,TS,TE>} \mid \exists x\ \exists y\ (\ x \in \text{X}_l \wedge y \in \text{Y}_l \\
& \wedge x.\text{S}=y.\text{S} \wedge y.\text{TE}<x.\text{TS} \\
& \wedge t.\text{S}=x.\text{S} \wedge t.\text{V}=x.\text{V} \wedge t.\text{U}=null \wedge t.\text{TS}=x.\text{TS} \wedge t.\text{TE}=x.\text{TE}\ )\ \}
\end{aligned}
$$

$$
\begin{aligned}
\text{TE}_o\text{-join}(\text{X},\text{Y}) \equiv \{\ t\text{<S,U,V,TS,TE>} \mid \exists x\ (\ & x \in \text{X} \\
& \wedge t.\text{S}=x.\text{S} \wedge t.\text{V}=x.\text{V} \wedge t.\text{U}=null \wedge t.\text{TS}=x.\text{TE} \\
& \wedge t.\text{TE}=x.\text{TE} \wedge \neg\ \exists y\ (\ y \in \text{Y} \wedge x.\text{S}=y.\text{S}\ )\ )\ \}
\end{aligned}
$$

Before we continue our discussion, let us emphasize once again that, to the best of our knowledge, all temporal join operators that have been proposed in the literature can be expressed in terms of conventional relational algebra. In other words, these join operators do not increase the expressiveness of the temporal query language (compared with the relational algebra). This argument is equally applicable in a later subsection which concerns snapshot operators.

3.3.2 "Within" Operator

This is a special kind of join operation and there are two "within" operators which are distinguished by the combination of operands: time intervals and time points. Given two time-interval relations X(S,U,TS,TE) and Y(S,V,TS,TE), we define the within-i-i operator as:

within-i-i(X,Y,N) holds if

- "[X.TS,X.TE) **intersect** [Y.TS,Y.TE)" holds, *or*
- "($0 \leq$ Y.TS$-$X.TE $<$ N) $\vee$ ($0 \leq$ X.TS$-$Y.TE $<$ N)" holds.

A pair of X and Y tuples satisfy the join condition if their "distance" does not exceed N. We note that this operator can be expressed in terms of SQL or QUEL queries. Essentially, using the operand relations (X and Y), one can obtain two temporary tables (denoted as X' and Y') — containing all the original tuples but with the TE value of each tuple incremented by N units of time. The within-i-i(X,Y,N) becomes intersect-join(X',Y'), i.e., a join between the two temporary tables.

Given a time-interval relation X(S,U,TS,TE) and a time-point relation Y(S,V, TA), we define the within-i-p operator as:

within-i-p(X,Y,N) holds if

- "Y.TA **between** [X.TS,X.TE)" holds, *or*
- "$0 \leq$ Y.TA$-$X.TE $<$ N" or "$0 \leq$ X.TS$-$Y.TA $\leq$ N" holds.

A pair of X and Y tuples satisfy the join condition if their "distance" does not exceed N. As in the case for the within-i-i operator, the within-i-p operator can be expressed in terms of SQL or QUEL queries. Given a time-interval temporal relation X, one can obtain a temporary table (denoted as X') — the TS value of each tuple is decremented by N units of time and the TE value is incremented by N units of time. Together with relation Y, the within-i-p(X,Y,N) operator becomes a join operation whose join condition is "X'.TS$\leq$ Y.TA $\wedge$ Y.TA$<X'$.TE".

3.3.3 Snapshot Operators

We discuss several commonly used snapshot operators (i.e., selection on timestamps) — between, intersect, as of, and time-slice operators whose use allows us to "view" the database content that is active during a particular time interval or at a particular time point.

The between, intersect, and as of operators can be defined in terms of comparison predicates on timestamps as follows:

- between — Given a time point T and a time interval $[t_s, t_e)$, "T between $[t_s, t_e)$" holds if and only if "$t_s \leq T \wedge T < t_e$" holds.

- intersect — Given two time intervals [TS,TE) and $[t_s, t_e)$, "[TS,TE) intersect $[t_s, t_e)$" holds if and only if "$t_s < TE \wedge TS < t_e$" holds. "$\sigma_P(R_1, \cdots, R_m)$ intersect $[t_s,t_e)$" is defined as:

  $$\sigma_{P \wedge \cdots \wedge [R_i.TS, R_i.TE) \text{ intersect } [t_s,t_e) \wedge \cdots} (R_1, \cdots, R_m)$$

  where P is a query qualification.

- as of — This operator is a special case of the intersect operator. Given a time interval [TS,TE) and a time point t, "[TS,TE) as of t" holds if and only if "t between [TS,TE)" holds. However, "[TS,TE) as of *now*" is equivalent to "TE=*now*". "$\sigma_P(R_1, \cdots, R_m)$ as of t" is defined as:

  $$\sigma_{P \wedge [R_1.TS, R_1.TE) \text{ as of } t \wedge \cdots \wedge [R_m.TS, R_m.TE) \text{ as of } t} (R_1, \cdots, R_m)$$

  where P is a query qualification.

In [Cli87], the time-slice operator is defined as the intersect operator except that its definition also requires that the lifespan of a selected tuple be the intersection of the lifespan of the qualified tuple and the query specific interval. As in the T-join that is discussed earlier, the intersection of the lifespans can be expressed in terms of a union of four different expressions. If we are only interested in selecting tuples whose lifespan intersects with the query-specific interval, the time-slice operator becomes a *single* conventional select operation:

$$\sigma_{P \wedge [X.TS, X.TE) \text{ intersect } [t_1,t_2)} (X).$$

In short, the snapshot operators are equivalent to a conjunction of several comparison predicates involving only timestamps. The following query, which is another example of select operation, selects tuples that satisfy a predicate P involving only non-time attributes during the entire interval $[t_1, t_2)$:

$$\sigma_{P \wedge X.TS \leq t_1 \wedge t_2 < X.TE} (X).$$

3.3.4 Time-project and Time-union Operators

The **time-project** operator, denoted as π_T, basically projects on the pair of timestamps of a temporal relation: $\pi_{X.TS,X.TE}$ (X). Together with a select operator, one can find the time intervals of tuples that satisfy a query qualification P:

$$\pi_T(\sigma_P (X)) = \pi_{X.TS,X.TE} (\sigma_P (X)).$$

We note that this combination of the **time-project** and select operators appears as the **tdom** operator in [Gad88] and as the dynamic **time-slice** operator in [Cli87]. For example, the following query retrieves from the time interval(s) during which Tom was the manager of Sales department:

$$\pi_T(\sigma_{Dname=Sales \wedge Mgr=Tom} (DEPT(Dname,Mgr,TS,TE))).$$

If a person was the manager of a department during several periods of time, more than one interval (not necessarily overlapping) may be returned. For example,

$$\pi_T(\sigma_{Mgr=Tom} (DEPT))$$

returns the interval(s) during which Tom was a manager. If Tom was the manager of several departments at the same time, the query response contains several tuples of which time intervals overlap. This leads to some observations. First, in the response to the query it is often more natural and intuitive to return one or more disjoint intervals each of which is equivalent to several overlapping or contiguous intervals. Towards this end, one can define a **time-union** operator which performs this operation. Second, the **time-union** operator can play a role in query optimization when the result from the **time-project** operator is joined with other temporal relations. However, the **time-union** operator is really a fixed point computation which cannot be expressed in terms of traditional relational algebra[9]. Essentially, the fixed point computation is to join the interval relation with itself repeatedly until no new tuple is generated. For an interval relation r(TS,TE), the join condition is the "overlap-join(r,r) or meet-join(r,r)". The following logic program (using syntax similar to Prolog [Ste86]) implements the time-union operator[10]:

```
time-union(TS,TE) :- concat(TS,TE), ¬ overlap(TS,TE).
concat(TS,TE) :- r(TS,TE).
concat(TS,Te) :- r(TS,TE), concat(Ts,Te), TS<Ts, Ts≤TE, TE<Te.
overlap(Ts,Te) :- r(TS,TE), TS<Ts, Ts≤TE.
overlap(Ts,Te) :- r(TS,TE), TS≤Te, Te<TE.
```

3.4 Classification of Queries

We first discuss a general classification of common queries and then focus on temporal join queries that are of interest in this dissertation.

3.4.1 General Classification

Common queries can be broadly classified using different criteria. Firstly, queries can be partitioned into select and join queries.

Select Query This is often viewed as n-dimensional range search, e.g., selecting employee records whose age is between 40 and 50: $\sigma_{40 \leq \text{age} \leq 50}$ (Employee). Various indexing techniques can be used to speed up the search process.

Join Query We consider two types of joins — *inequality join* and *equi-join*. Inequality joins are often expensive to process while equi-joins can be processed efficiently by a number of algorithms such as sort-merge joins and hash joins.

Secondly, we can partition attributes in the query qualification into 1) non-time attributes only and 2) timestamps only. Together with the select and join

[9] Incidentally, a variant of this fixed point computation is proposed as a linear recursion operator in [Tuz90]. Their data model, however, only implicitly references timestamps.

[10] Unfortunately there is no "standard" language or operator for recursion and for this reason, we use a logic programming language. On the other hand, one need not implement the time-union operator using recursions [Leu91].

query classification, we obtain a 2 by 3 matrix:

	non-time attributes only	timestamps only
select	$\sigma_{V \geq 50}(R_1)$	$\sigma_{TS<50}(R_1)$
equi-join	$\sigma_{R_1.U=R_2.V}(R_1, R_2)$	$\sigma_{R_1.TS=R_2.TS}(R_1, R_2)$
inequality join	$\sigma_{R_1.U<R_2.V}(R_1, R_2)$	$\sigma_{R_1.TE<R_2.TS}(R_1, R_2)$

where R_1 and R_2 are time-interval temporal relations. Note that a query can contain a mixture of these qualifications.

Thirdly, any query in these six categories can further be augmented with a snapshot operator, forming a snapshot or interval query. We distinguish two types of snapshot (or similarly interval) queries: *snapshot select* and *snapshot join* queries. The first type involves selection based on attribute values as of a certain time: e.g., $\sigma_{S=s1}(R)$ as of t_i while the second type involves joining several relations as of a certain time: e.g., $\sigma_P(R_1, \cdots, R_m)$ as of t_i. Since snapshot operators also involve timestamps, queries with both a snapshot (especially as of) operator and other qualification on timestamps may produce null responses if the "combined" query qualifications evaluates to false. For example, the following queries return null responses:

- $\sigma_{R.TE<t}(R)$ as of $t \equiv \sigma_{R.TE<t \wedge R.TS \leq t \wedge t<R.TE}(R)$
- $\sigma_{R_1.TE<R_2.TS}(R_1, R_2)$ as of t
  $\equiv \sigma_{R_1.TE<R_2.TS \wedge R_1.TS \leq t \wedge t<R_1.TE \wedge R_2.TS \leq t \wedge t<R_2.TE}(R_1, R_2)$

3.4.2 Temporal Select-Join Queries

We define a general class of queries called Temporal Select-Join (TSJ) as follows.

Definition 3.3 TSJ is defined as the set of relational algebraic expressions of the following form:

$$\sigma_{P(R_1,\cdots,R_m)}(R_1 \times \cdots \times R_m) \qquad \text{or} \qquad \sigma_{P(R_1,\cdots,R_m)}(R_1, \cdots, R_m)$$

where $P(R_1, \cdots, R_m)$ or simply P is a query qualification and $R_1, \cdots, R_m$, $m \geq 1$, are time-interval temporal relations. □

We distinguish between several subclasses of TSJ join queries. This classification provides a meaningful partitioning of query types with respect to the difficulty and complexity of query processing and optimization. Each subclass has a restricted form of query qualification and is amenable to a particular query processing algorithm. Informally, their characterizations are:

Disjoint Join The join condition between two tuples does not require that their intervals overlap, as illustrated in Figure 3.4(a). For example, queries with join conditions "R_i.TE<R_j.TS" or "R_i.TE<R_j.TE" belong to this category.

Overlap Join The join condition between two tuples requires that their intervals share a common time point, i.e., they overlap. We consider two special kinds of overlap joins whose formal definitions will be presented shortly:

TSJ$_1$ — All participating tuples that satisfy the join condition share a common time point, as illustrated in Figure 3.4(b). For example, finding a complex "event pattern" in which all events occur during the same period of time (or as of a particular time point) can be viewed as a TSJ$_1$ join query.

TSJ$_2$ — The tuples that satisfy the join condition overlap in a "chain" fashion, as illustrated in Figure 3.4(c). However, not all participating tuples that satisfy the join condition share a common time point. For example, finding an event pattern in which events occur in some overlapping sequence can be viewed as a TSJ$_2$ join query.

Note that all TSJ$_1$ queries are also TSJ$_2$ queries.

We now precisely define the classes of queries that are of interest here. Given a query $Q \equiv \sigma_{P(R_1,\cdots,R_m)}(R_1,\cdots,R_m)$, we construct a join graph (denoted as G) from the query qualification $P(R_1,\cdots,R_m)$ using Algorithm 3.1 below. Based on the join graph, we are able to formally define TSJ$_1$ and TSJ$_2$ join queries.

Algorithm 3.1 Join Graph.

There are m nodes in the join graph G; each node represents an operand relation R_i, $1 \leq i \leq m$, and is labeled with the name of that relation. We add an undirected edge between nodes R_i and R_j ($i \neq j$) to G if the following condition is satisfied:

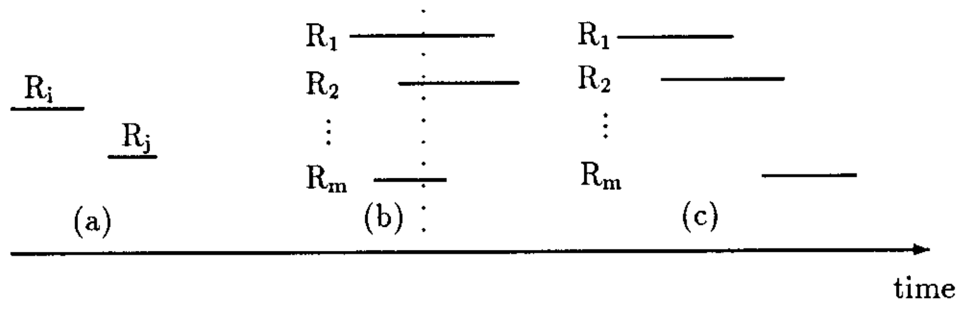


Figure 3.4: Classes of temporal joins

$$P(R_1, \cdots, R_m) \Rightarrow R_i.TS \leq R_j.TE \wedge R_j.TS \leq R_i.TE.$$

That is, for each m-tuple $<r_1, \cdots, r_m>$, where $r_k \in R_k$, $1 \leq k \leq m$, that satisfies the qualification $P(R_1, \cdots, R_m)$, r_i and r_j must span a common time point[11]. □

Definition 3.4 TSJ$_2$: A query $Q \equiv \sigma_{P(R_1,\cdots,R_m)} (R_1, \cdots, R_m)$ belongs to TSJ$_2$ if the following conditions hold:

1. The number of operand relations in Q is greater than 1, i.e., m>1.
2. The join graph G constructed using Algorithm 3.1 is a connected graph, i.e., all nodes in G are connected.

□

Definition 3.5 TSJ$_1$: A TSJ$_2$ query is also a TSJ$_1$ query if the join graph G constructed using Algorithm 3.1 is a *fully connected* graph. In other words, for all i and j such that i, j $\in \{1,\cdots,m\}$ and $i \neq j$:

$$P(R_1, \cdots, R_m) \Rightarrow R_i.TS \leq R_j.TE \wedge R_j.TS \leq R_i.TE.$$

[11] This condition is defined such that we can also handle the join predicate "X.TE=Y.TS" for a join of two relations. Testing the implications can be readily achieved via algorithms presented in [Ros80, Ull82, Sun89]. Moreover, semantic constraints optimization can be used to add more edges in the graph.

That is, for each m-tuple $<r_1, \cdots, r_m>$, where $r_k \in R_k$, $1 \le k \le m$, that satisfies the join condition $P(R_1, \cdots, R_m)$, all participating tuples (r_k's) must span a common time point. □

Theorem 3.1 Common Time Point:

Given a query $Q \equiv \sigma_{P(R_1,\cdots,R_m)} (R_1, \cdots, R_m) \in TSJ_1$. For each m-tuple $<r_1, \cdots, r_m>$, where $r_k \in R_k$, $1 \le k \le m$, that satisfies the join condition P, all participating tuples (r_k's) span a common time point. □

Proof:

The proof basically follows from the definition of TSJ_1. Consider a m-tuple $<r_1, \cdots, r_m>$, where $r_k \in R_k$, $1 \le k \le m$, that satisfies the join condition $P(R_1, \cdots, R_m)$. Without loss of generality, let us assume that tuples r_i and r_j do not share a common time point, where i, j $\in \{1, \cdots, m\}$ and $i \ne j$. Then, exactly one of the following conditions holds:

1. r_i.TE < r_j.TS, i.e., r_i ends before r_j starts.
2. r_j.TE < r_i.TS, i.e., r_j ends before r_i starts.

In either case, the m-tuple $<r_1, \cdots, r_m>$ would not have been generated by the join because "r_i.TS≤r_j.TE ∧ r_j.TS≤r_i.TE" is false and thus the m-tuple does not satisfy the join condition $P(R_1, \cdots, R_m)$. Q.E.D.

The class of TSJ_1 and TSJ_2 are multi-way temporal joins (such as temporal pattern queries) in which the lifespans of tuples intersect. The main characteristic of these join conditions is that a tuple $r_i \in R_i$ does not join with (theoretically) infinitely many "future" tuples $r_j \in R_j$ which start after r_i ends[12]. Counter-examples include the Cartesian products across multiple relations (i.e., no join

[12] Or conversely, a tuple $r_j \in R_j$ which starts at time t does not join with (theoretically) infinitely many "past" tuples $r_i \in R_i$ which ends before r_j starts.

predicates) and a query with the join condition "R_i.TE<R_j.TS". This characteristic is crucial in developing the generalized data stream index and the parallel query processing schemes to be described in later chapters.

Examples of TSJ$_1$ query include the commonly called "natural time join" [Cli85, Cli87], "intersection joins" [Gun91] which join relations in first temporal normal form [Seg88]. Also, the temporal join operators listed in Figure 3.2 belong to TSJ$_1$.

Example 3.1 Given temporal relations X, Y and Z. The following is a TSJ$_1$ query:

$$\sigma_{\text{contain-join(X,Y)} \wedge \text{contain-join(Y,Z)}} (X,Y,Z).$$

Although there is no explicit join predicate between relations X and Z, the TSJ$_1$ definition is still satisfied. However, the following is a TSJ$_2$ query but not TSJ$_1$:

$$\sigma_{\text{intersect-join(X,Y)} \wedge \text{intersect-join(Y,Z)}} (X,Y,Z).$$

□

In Section 6.4, we will discuss TSJ$_2$ join queries in more detail. Until then, we will focus on TSJ$_1$ queries. In the next subsection, we show that common temporal queries can often be formulated as TSJ$_1$ queries.

3.4.3 Example TSJ$_1$ Queries

Consider the following temporal relations which store information on studios, directors and stars in the film industry[13]:

Studio(Sname,Head,TS,TE) — the head of a studio
Dir(Dname,Sname,TS,TE) — the director who worked for a studio
Stars(Star,Dname,TS,TE) — the actor/actress in a film directed by a director

where Sname and Dname stand for the name of studios and directors respectively.

[13] These relations are adopted from the examples in [Cli87].

Example 3.2 Find the head of a studio and the director who worked for the studio at the same time:

$\sigma_{\text{intersect-join(Studio,Dir)} \wedge \text{Studio.Sname=Dir.Sname}}$ (Studio,Dir).[14]

□

Example 3.3 Find all combinations of studio heads, film stars and directors such that the studio head was a star in a film that the director directed at the studio during the same period of time:

$\sigma_{P_1 \wedge P_2}$ (Studio,Dir,Stars)

where P_1 is "Studio.Sname=Dir.Sname ∧ Studio.Head=Stars.Star ∧ Dir.Dname =Stars.Dname" and P_2 is "intersect-join(Studio,Dir) ∧ intersect-join(Studio, Stars) ∧ intersect-join(Dir,Stars)". □

Let us consider examples in a different application domain. Suppose we have the following temporal relations which store information regarding departments and employees:

Emp(Ename,Salary,TS,TE) — employee salary
Dept(Dname,Mgr,TS,TE) — department manager
Sales(Dname,Revenue,TS,TE) — department sale revenue

where Ename and Dname stand for the name of employees and departments respectively, and Mgr stands for the name of the department manager.

Example 3.4 Find the manager whose department revenue was less than \$200K:

$\sigma_{\text{intersect-join(Dept,Sales)} \wedge \text{Sales.Revenue<\$200K}}$ (Dept,Sales).

□

[14] A more appropriate response might include two "computed" fields which represent the lifespan of a joined tuple. In this dissertation, we focus only on the query qualification which is a major optimization issue.

Example 3.5 "Inefficient management": suppose we want to find the combination of departments and employees such that during the period of time when the employee earned more than \$100K, the employee was the manager of the department, and during the same period of time the department's revenue was less than \$500K. This query can be formulated as:

$$\sigma_{\text{contain-join(Sales,Emp)} \wedge \text{contain-join(Dept,Emp)} \wedge \text{P}} (\text{Emp,Dept,Sales})$$

where P is "Emp.Salary>\$100K ∧ Sales.Revenue<\$500K ∧ Sales.Dname=Dept.Dname ∧ Emp.Ename=Dept.Mgr". □

In the following chapters, we discuss the optimization issues raised by the use of various temporal operators. First, we present a *stream processing* approach to processing temporal join and semijoin operations. As temporal data often has certain implicit ordering by time, the stream processing approach which takes advantage of data ordering is often the preferable alternative to conventional methods.

CHAPTER 4

Processing Temporal Join Operators

In this chapter, we consider query processing and optimization for temporal join queries, a topic which receives little attention until recently [Seg89, Leu90, Gun91]. We observe that there are several interesting characteristics which are peculiar to temporal databases: (1) a temporal join query often involves patterns of events; (2) a temporal join query often contains a conjunction of several inequalities over the time domain and no equality conditions; and (3) temporal data is rich in semantics, and semantic query optimization is particularly desirable in the presence of a number of inequalities. These characteristics provide new opportunities for optimization. Ignoring these, as in conventional relational systems, can result in poor performance.

We discuss join and semijoin operations which are the most common and expensive computations in database systems. We introduce a stream processing approach which takes advantage of data ordering. As temporal data often has certain implicit ordering by time, the stream processing approach, as we demonstrate, is a good alternative. We should emphasize, however, that the stream processing algorithms that we present are merely additional strategies that a query optimizer should consider, and are by no means substitutes for traditional query processing methods such as the nested-loop joins.

The idea of stream processing has also appeared in [Ben79, Pre85, Ore88, Par90]. These researches share the basic principle of the stream processing paradigm in that input data should be in a certain order before the processing commences. [Ore88] focuses on processing spatial image data stored using pixel representation which is not appropriate in many proposed temporal data models. Its implementation relies on a special transformation, called the z-transform, and consequently there is only one interesting ordering, namely the z-order (lexicographical ordering of z-transformed values). In [Ben79, Pre85], a main memory algorithm, called the plane sweep algorithm, is discussed and can be used to report all intersecting pairs of planar line segments. Although time attributes (i.e., TS and TE timestamps) can be thought of as the endpoints of horizontal line segments, the algorithm is primarily designed to find all line segment pairs which intersects at a single point, and therefore it is not directly applicable to processing join queries. Moreover, only a single sort ordering of data items is considered in this algorithm while in our approaches, temporal relations can be sorted on the TS or TE timestamp. Nonetheless, these methods emphasize the importance of data ordering. Here we are more concerned with (1) the impact of various data ordering on performance issues, mainly memory workspace requirements, and (2) efficient processing algorithms for temporal join and semijoin operations. As we show, the optimal sort ordering for these temporal join operators may depend on the statistics of data instances as well as the operator itself.

One may argue that stream processing algorithms are not useful unless data is properly sorted. With an abundant amount of main memory and processing cycles available, one can sort the input streams "on the fly" with marginal additional cost as assumed in [Sto88]. The algorithms we describe would still be applicable although the streams are actually memory resident. Furthermore, as we described in Chapter 3, there are natural situations where tuples are sorted on TS or TE timestamp when the append-only update policy is in use.

Semantic query optimization has been discussed in the literature [Kin81, Chak84, Jar84, She89] but apparently has not been widely used in conventional systems. Semantic constraints in temporal databases occur more naturally and are more plentiful, and consequently a query optimizer should profitably exploit the semantics. We will briefly discuss the role of semantic query optimization in temporal databases after discussing more basic query processing and optimization issues.

The remainder of this chapter is organized as follows. We illustrate, in Section 4.1, the conventional approach to processing a complex temporal query. In Section 4.2, we discuss a stream processing approach for the implementation of temporal join operators. We informally discuss the role of semantic query optimization in Section 4.3. A brief conclusion is included in Section 4.4.

4.1 Conventional Approach

In this section, we describe the deficiencies of conventional relational database systems in processing temporal queries. Temporal queries using the extended constructs (i.e., temporal operators) described in the previous chapter are usually processed in the following way. First, queries with temporal operators are translated into equivalent queries in a relational language such as Quel. The translated queries are then optimized and processed by conventional relational query processors. This approach is generally inefficient for processing temporal queries as we demonstrate below.

Consider a relation Faculty(Name,Rank,TS,TE) as described in the previous chapter, and the following Quel query modified from [Sno87][1]: *Superstar — Who got promoted from assistant to full professor while at least one other faculty remained at the associate rank?*

```
range of f1 is Faculty
range of f2 is Faculty
range of f3 is Faculty
retrieve into Stars(Name=f1.Name,TS=f1.TE,TE=f2.TS)
where f3.Rank=associate and f1.Name=f2.Name and
        f1.Rank=assistant and f2.Rank=full and
        (f1 intersect f3) and (f2 intersect f3)
```

These intersect operators can be translated directly into equivalent clauses (e.g., in Quel) involving inequalities. That is,

$$(\text{f1 intersect f3}) \equiv \text{f1.TS}<\text{f3.TE} \wedge \text{f3.TS}<\text{f1.TE}$$
$$(\text{f2 intersect f3}) \equiv \text{f2.TS}<\text{f3.TE} \wedge \text{f3.TS}<\text{f2.TE}$$

[1] The original TQuel query in [Sno87] is:

```
range of f1 is Faculty
range of f2 is Faculty
range of a is Associate
retrieve into Stars(Name=f1.Name)
        valid from begin of f1 to begin of f2
        where f1.Name=f2.Name and f1.Rank=assistant and f2.Rank=full
        when (f1 overlap a) and (f2 overlap a)
```

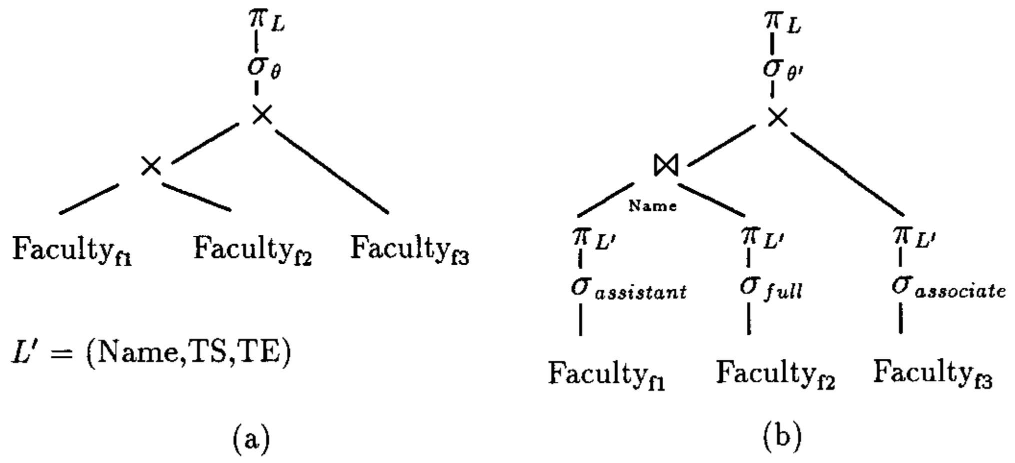


Figure 4.1: (a) Parse tree for the Superstar expression and (b) its optimized version

The corresponding relational algebraic expression for the Superstar query is:

$$\pi_L\,(\sigma_\theta\,(\text{Faculty}_{f1} \times \text{Faculty}_{f2} \times \text{Faculty}_{f3}))$$

$$\begin{array}{rll} \text{where } L & \text{is} & \text{f1.Name, f1.TE, f2.TS} \\ \theta & \text{is} & \text{f1.Name=f2.Name} \wedge \text{f1.Rank=assistant} \wedge \\ & & \text{f2.Rank=full} \wedge \text{f3.Rank=associate} \wedge \theta' \\ \theta' & \text{is} & \text{f1.TS<f3.TE} \wedge \text{f3.TS<f1.TE} \wedge \\ & & \text{f2.TS<f3.TE} \wedge \text{f3.TS<f2.TE} \end{array}$$

This algebraic expression can be represented as a parse tree [Ull82], as depicted in Figure 4.1(a). The parse tree can then be ameliorated by applying well-known traditional algebraic manipulation methods, e.g., the selections and projection are pushed as far down the parse tree as possible (see Figure 4.1(b)).

There are several interesting observations about the "conventionally optimized" parse tree in Figure 4.1(b):

1. There are three references to the Faculty relation in the parse tree implying that it is joined with itself twice — conventional systems would scan the relation several times. If we view the query as a "Superstar" *pattern*

matching against the Faculty relation, one might wonder if we are able to answer this query with only a single scan of the relation. Roughly speaking, we are looking for a pattern composed of three tuples — an assistant professor, a full professor and an associate professor. That is, instead of performing multiple scans, a single scan of the relation might be possible by recognizing this query qualification as describing a pattern against the data.

2. The first join (i.e., "f1.Name=f2.Name") in the parse tree can be efficiently implemented as an equi-join using conventional approaches such as nested-loop join, merge join or hash join. The second join, an *inequality join*, is a Cartesian product followed by a selection with the condition being a conjunction of inequality predicates — θ' [2]. Traditionally, the best strategy for processing inequality joins appears to be the conventional nested-loop join method[3]. With only a single inequality as the join condition, we have no choice but the nested-loop join method. Since time points are totally ordered and the join condition is a conjunction of several inequalities involving timestamps, one might wonder if there are any more efficient processing alternatives. In the past, little attention has been given to this form of query qualification because:

   - in traditional database applications, queries seldom contain inequality joins, and
   - when inequalities do occur, in most situations the join condition has only a single inequality predicate; for example, in a database which stores employee and department relations, we might want to retrieve employees who earn more than their manager.

   The situation is quite different when we consider temporal databases:

   - inequality joins appear more frequently and naturally because temporal queries often involve patterns of events, and therefore inequality joins need to be explicitly considered in the query optimization,

[2] Note that range search (e.g., salary $>$ 10K and salary $<$ 20K) is different from this form of query qualification.

[3] One can do a little better in that the inner loop may terminate even before finishing the scanning of the relation. However, this "modified" nested-loop join still requires multiple scans of operands.

- the join condition often contains a conjunction of several inequality predicates which further indicates that optimization might be possible.

3. Recall that there is an integrity constraint in the Faculty relation: a chronological ordering of data values — assistant, associate and full. This ordering implies that being an assistant professor must occur before being promoted to a full professor, i.e., "f1.TE<f2.TS" always holds in the presence of "f1.Name=f2.Name". This constraint, together with the "intra-tuple" integrity constraints,

   fi.TS<fi.TE   for i=1,2,3

   imply both "f1.TS<f3.TE" and "f3.TS<f2.TE" hold. Therefore these inequalities in θ' are redundant — i.e., they are subsumed by other inequalities. The important point is not so much this particular case; rather it is the process of semantic query optimization.

The above observations suggest that, in addition to traditional set-oriented relational operators, we may need other alternatives to process temporal queries. In subsequent sections we will present and discuss a number of such alternatives.

4.2 The New Approach

We discuss a *stream processing* approach for temporal query processing in this section. Algorithms that implement temporal operators are presented. The tradeoffs among sort orders, the amount of local workspace and multiple passes over input streams are discussed. For properly sorted streams of tuples, we show that temporal join operators can often be carried out with a single pass of input streams and the amount of workspace required can be small.

4.2.1 The Notion of Stream Processing

Abstractly, a *stream* can be defined as an ordered sequence of data objects [Abe85, Par89, Par90]. Stream processing resembles the notion of dataflow processing in database systems [Bor82, Bat88, Ore88]. A classical example of stream processing is the merge-join where both operands are sorted on the join attributes; the join can be efficiently executed and the sort ordering of its output can be utilized by subsequent operations [Smi75, Sel79].

There are several intrinsic characteristics of stream processing in database systems. First, a computation on a stream has access only to one element at a time and only in the specified ordering of the stream (via a *data stream pointer*). Second, the implementation of a function as a stream processor may keep some local state information in order to avoid multiple readings of data streams. The state information represents a summary of the history of a computation on the portion of a stream that has been read so far; the state may be composed of copies of some objects or some summary information of the objects previously read (e.g., sum, min, count etc.) Using the local state information, the implementation of a stream processor can be expressed in terms of functions on the individual objects at the head of each input stream and the current state. Third, there are often tradeoffs among the following factors:

1. the minimal size of the local workspace which depends on the function itself and the statistics of specific instances of data streams,
2. the sort ordering of input data streams, and
3. multiple passes over input data streams.

Very often stream processing requires input streams to be properly sorted in order to perform the computation while only reading the input streams once. In addition, the sort orderings of input streams greatly affect the size of local workspace required. Conversely, suppose there is enough local workspace to keep all data objects. Then only a single pass over the input streams is required and theoretically the sort ordering would not be important.

Let us consider a simple stream processor, which is shown in Figure 4.2, lists all employees and computes their average salary. The input is a data stream of employee salary tuples — the period when an employee earned a particular salary, and the output is a data stream of employee and his/her averaged salary. The point here is that the state contains summary information (i.e., the partial sum and length of time interval), and the function (i.e., average) is expressed in terms of the current state and an input tuple. Suppose that the input data stream is sorted or clustered on the employee name. The stream processor needs

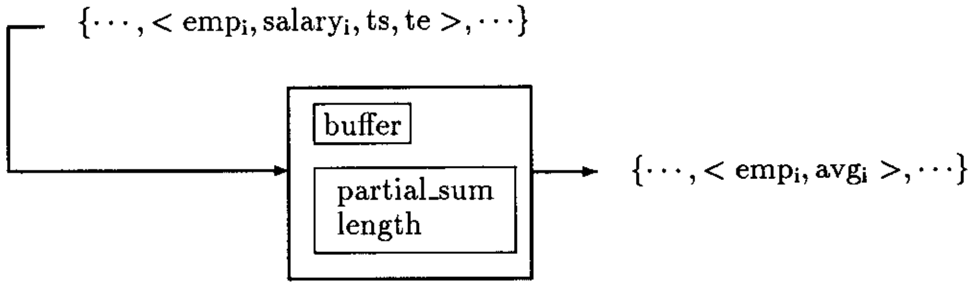

$\{\cdots, < \mathrm{emp_i}, \mathrm{salary_i}, \mathrm{ts}, \mathrm{te} >, \cdots\}$
buffer
partial_sum
length
$\{\cdots, < \mathrm{emp_i}, \mathrm{avg_i} >, \cdots\}$

Figure 4.2: A stream processor which computes the average salary for each employee

only one set of registers for the partial sum and the length of employment as state information. As soon as a record of new employee is read from the data stream, the stream processor can output the result for the previous employee. That is, the input data stream is read only once. Now suppose the data stream is neither sorted nor clustered on the employee name, the stream processor would have to keep a table of registers — a set of registers for each employee. That is, more workspace is required so that the input data stream is read only once. If there is not enough workspace for the entire table of registers (i.e., we keep only a portion of the table), the input data stream have to be read multiple times for the computation. These are the tradeoffs for this example.

In the next section, we discuss the application of stream processing techniques to processing temporal joins. In these discussions, the sort ordering of streams plays a major role.

4.2.2 Sort Orderings

Suppose we have temporal relations X(S,U,TS,TE) and Y(S,V,TS,TE). We are interested in the effect of various sort orderings on the efficiency with which it is possible to implement the temporal join operators (listed in Figure 3.2) in the stream processing paradigm. We illustrate the idea using the "contain" relationship which has only inequalities in its explicit constraints[4]. Before we

[4] For temporal operator with equality predicate(s), an obvious stream processing method appears to be sorting both relations on attributes that are involved in the equalities followed by a conventional merge-join, and perhaps combined with filtering using other predicates in the

proceed, we should note that the operators listed in Figure 3.2 are in fact join and semijoin operations. Because of this, the only form of state information we need consider is subsets of the tuples previously read and not any summary information such as sum, min, etc.

4.2.2.1 Contain-join(X,Y)

The contain-join(X,Y) outputs the concatenation of tuples X and Y if the lifespan of X contains that of Y; that is, "X.TS<Y.TS ∧ Y.TE<X.TE" — i.e., the "contain" relationship in Figure 3.2. The generic algorithm for processing a temporal join operator (such as inequality join) is shown in Figure 4.3. The specific instance of this generic algorithm depends on the sort orderings of data streams. In this chapter, we present the contain-join algorithm in more detail for the case when both relations X and Y are sorted on the TS timestamp in ascending order (see Figure 4.4(a)). The following conventions and notations are used in the algorithm:

1. There is an input buffer for reading tuples from each stream, denoted as <Buffer-x, Buffer-y>, and the tuples in these buffers are denoted as x_b and y_b respectively.

2. The expected difference between TS values of two consecutive X (respectively Y) tuples is τ_x (respectively τ_y). If there is no selection on the input data stream, τ_x is T_{ins} shown in Figure 3.1.

3. The absolute value of the difference between y_b.TS and x_b.TS is denoted as l. That is the "distance" between the data stream pointers of X and Y.

Algorithm 4.1 Contain-join(X,Y).[5]

1. Initially the first tuple from each stream is read and stored in the buffer.

operator.

[5] The separation of this join algorithm into several phases is primarily for the sake of explanation; it is possible that Steps 2, 3 and 4 can be combined together to gain better performance.

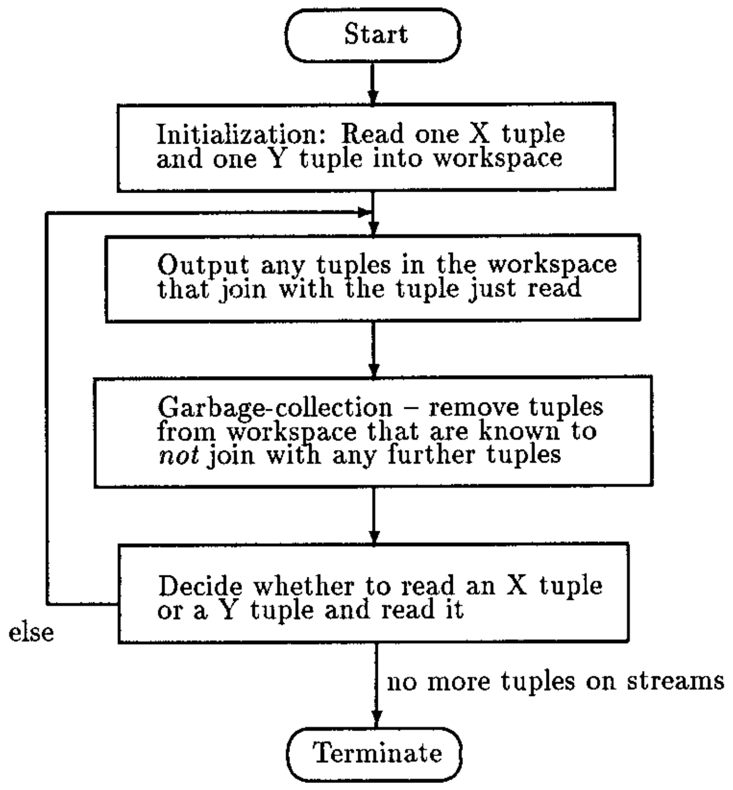

Start
Initialization: Read one X tuple and one Y tuple into workspace
Output any tuples in the workspace that join with the tuple just read
Garbage-collection – remove tuples from workspace that are known to *not* join with any further tuples
Decide whether to read an X tuple or a Y tuple and read it
else
no more tuples on streams
Terminate

Figure 4.3: Generic stream processing algorithm for temporal joins

2. Join phase: Output the tuple pair for any tuples in the workspace (i.e., the state information) that join with the tuple just read. Note that additional housekeeping must be done for maintaining the output sort ordering.

3. Garbage-collection phase: discard X tuples in the workspace if "X.TE$\leq y_b$.TS" holds. Also discard Y tuples if "Y.TS$\leq x_b$.TS" holds. The garbage-collection conditions must guarantee that the Y (respectively X) tuples being discarded do not satisfy the join condition with any subsequent X (respectively Y) tuples that have not been read.

4. Read phase: Copy the previously read tuple(s) into the workspace as state tuple(s). There are two different situations in deciding which stream of tuples is to be read. The first case is when "y_b.TS$\leq x_b$.TS" as shown in Figure 4.4(b). As all Y tuples read so far do not join with x_b, clearly the next step is to read the next Y tuple. The second case is when "y_b.TS$>x_b$.TS" as shown in Figure 4.4(c). The workspace contains:
   (1) X tuples whose lifespan span y_b.TS and
   (2) Y tuples whose TS value is in region l.
   A heuristic can be used to decide whether to read the next X tuple or Y tuple which is presented below.

5. The algorithm terminates if either stream has been exhausted and there is no corresponding state tuple. Otherwise, go to Step 2. □

A heuristic algorithm which decides whether to read an X tuple or a Y tuple is as follows. If the next X tuple is read, the expected TS value is x_b.TS$+\tau_x$. The number of Y tuples that would be garbage-collected can be estimated as the number of Y tuples in the workspace with TS value in the interval [x_b.TS, x_b.TS$+\tau_x$]. If the next Y tuple is read, the expected TS value is y_b.TS$+\tau_y$. The number of disposable X tuples can be estimated as the number of X tuples in the workspace with TE value in the interval [y_b.TS, y_b.TS$+\tau_y$]. Based on these two estimations, a decision can be made on which would yield a greater reduction in the number of tuples in the workspace.

Another heuristic algorithm is this: if the number of Y (respectively X) tuples in the workspace is larger than the number of X (respectively Y) tuples, the next X (respectively Y) tuple will be read. If it is a tie, a tuple from either data stream

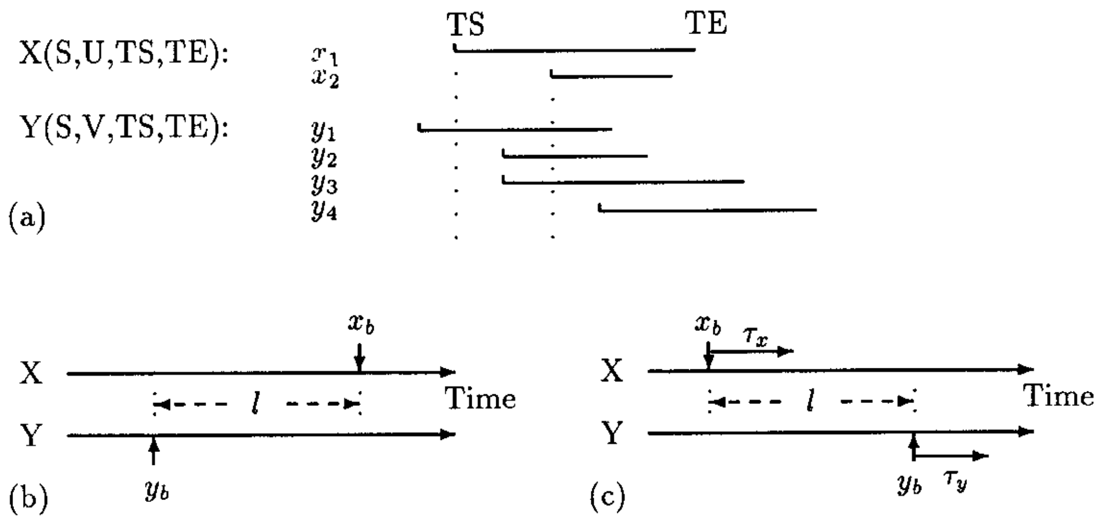


Figure 4.4: Both X and Y are sorted on TS in ascending order (only the lifespans are shown)

can be read. The relative merit of this simple heuristic is that it does not depend on the data statistics (i.e., τ_x and τ_y). However, the expected performance may not be as well as the previous heuristic algorithm.

For the contain-join operator, the contents of the workspace can be characterized as follows:

1. If we keep reading X tuples such that all Y tuples in the state have been garbage-collected, the maximal set of X tuples that are required consists of all X tuples that span the time instant y_b.TS.
2. Conversely, if we keep reading Y tuples such that there is no X state tuple, the maximal set of Y state tuples that is required consists of those whose TS value lie in the lifespan of x_b.

For the case when the relation X is sorted on TS and the relation Y is sorted on TE in ascending order, the algorithm is similar to the above one with the following exceptions:

1. Read phase: In the first heuristic algorithm which uses data statistics, if the next Y tuple is read, the expected disposable X tuples are those in the

interval [y_b.TE,y_b.TE+τ_y'] where τ_y' is the expected difference between the TE values of 2 consecutive Y tuples.

2. Garbage-collection phase: Dispose of X tuples if "X.TE$\leq y_b$.TE", and dispose of Y tuples if "Y.TS$\leq x_b$.TS".

3. The state is {X tuples whose lifespan span y_b.TE} $\cup$ {Y tuples whose lifespans are contained within l}.

We now summarize the state information requirements of processing the contain-join for other sort orderings in Table 4.1. Note that (1) it is generally inappropriate to have one relation sorted in ascending order and the other in descending order. (2) Sorting both relations X and Y on attribute TE in descending order would have the same effect as sorting them on attribute TS in ascending order because of symmetry (although the TS and TE timestamps "exchange" their roles); the lower half of Table 4.1 is therefore the mirror image of the upper half.

4.2.2.2 Contained-semijoin(X,Y) & Contain-semijoin(X,Y)[6]

Contain-semijoin(X,Y) is defined as $\{x \mid x \in X$ and $\exists\, y \in Y$ such that x's lifespan contains y's lifespan$\}$. Contained-semijoin(X,Y) is defined as $\{x \mid x \in X$ and $\exists\, y \in Y$ such that x's lifespan is contained in y's lifespan$\}$. In a later section, we show that contained-semijoin may be used to efficiently process the Superstar query.

For semijoins, a stream processor can output a tuple as soon as it finds the first matching tuple. Based on this property, we devise an efficient algorithm which requires just one buffer for each input stream for the case when relation X is sorted on TS and relation Y is sorted on TE in ascending order as shown in Figure 4.5. The contain-semijoin(X,Y) algorithm for this sort ordering is shown as follows.

Algorithm 4.2 Contain-semijoin(X,Y).

[6] Similar to "restriction" operator in [Seg87].

sort orders				contain	contain	contained
relation X		relation Y		-join(X,Y)	-semijoin(X,Y)	-semijoin(X,Y)
TS	↑	TS	↑	(a)	(c)	(c)
TS	↓	TS	↓	-	-	-
TS	↑	TE	↑	(b)	(d)	-
TS	↓	TE	↓	-	-	(d)
TE	↑	TS	↑	-	-	(d)
TE	↓	TS	↓	(b)	(d)	-
TE	↑	TE	↑	-	-	-
TE	↓	TE	↓	(a)	(c)	(c)

↑ Sorting the corresponding timestamp in ascending order.

↓ Sorting the corresponding timestamp in descending order.

- The sort ordering is not appropriate for stream processing – no garbage-collection criteria.

(a) state = {X tuples whose lifespan span y_b.TS}
∪ {Y tuples whose TS value lie in region l}

(b) state = {X tuples whose lifespan span y_b.TE}
∪ {Y tuples whose lifespans are contained within region l}

(c) state ⊆ {X tuples whose lifespan span y_b.TS}
∪ {Y tuples whose TS values lie in region l}

(d) local workspace = <Buffer-x, Buffer-y>.

Table 4.1: Effect of various sort orders on contain-join, contain-semijoin & contained-semijoin

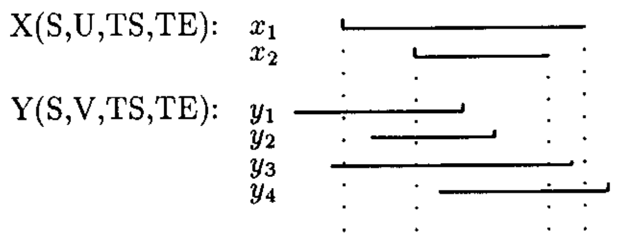


Figure 4.5: X is sorted on TS and Y is sorted on TE in ascending order

Repeat the following steps until one data stream is exhausted. Suppose we have x_b and y_b tuples in the input buffer. Then, one of the following conditions must hold:

- "x_b.TS<y_b.TS $\wedge$ y_b.TE<x_b.TE" — i.e., x_b and y_b satisfy the semijoin condition, and thus x_b is output. The next X tuple is read.
- "y_b.TS$\leq x_b$.TS" — i.e., x_b and y_b do not satisfy the semijoin condition[7]. Furthermore, y_b cannot be contained in subsequent X tuples which have larger TS values, and thus the next Y tuple is read.
- "x_b.TE$\leq y_b$.TE" — i.e., x_b and y_b do not satisfy the semijoin. Furthermore, x_b cannot be joined with subsequent Y tuples which have larger TE values, and thus the next X tuple is read.

It can be easily verified that for this particular data sort orderings only one tuple from each stream needs to be kept in the main memory. □

It should be mentioned that the above algorithm can also be applied to contained-semijoin(Y,X) for the same sort orderings with slight modification — when the semijoin condition is satisfied, y_b is output and the next Y tuple is read. For other sort orderings (e.g., both streams are sorted on TS), we only list the local workspace requirements in Table 4.1.

[7] It does not matter what the relationship between x_b.TE and y_b.TE is.

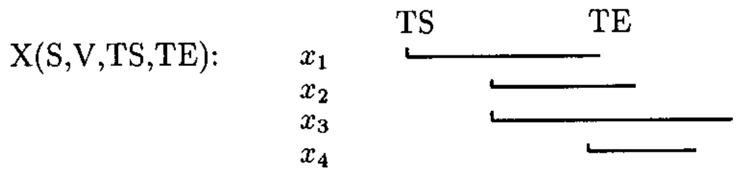


Figure 4.6: Relation X is sorted on TS (and then TE) in ascending order

sort order on X		contained-semijoin(X,X)	contain-semijoin(X,X)
TS	$\uparrow$	(a)	(b)
TS	$\downarrow$	-	(a)

(a) the state is $\{x_s\}$ and Buffer-x for x_b.

(b) if X is sorted into $\{x_1, \cdots, x_n\}$, the state for a tuple x_i is a subset of:
$\{x_j \mid \text{j} > \text{i and } x_j \text{ overlaps with } x_i\}$.

Table 4.2: Effect of various sort orders on the contained-semijoin(X,X) and contain-semijoin(X,X)

We note that for contain-semijoin(X,X) and contained-semi-join(X,X), the stream of tuples may be scanned twice if we apply the semijoin algorithm presented above. To avoid this kind of inefficiency, we therefore devise a more efficient algorithm which scans the stream only once provided that it is sorted properly. As an example, suppose the relation X has primary sort ordering on TS and secondary sort ordering on TE in ascending order as shown in Figure 4.6, only two buffers are required (see Table 4.2 for summary). The algorithm for contained-semijoin(X,X) is as follows, and in the next section, we discuss the circumstances under which this algorithm can be used for the Superstar query.

Algorithm 4.3 Contained-semijoin(X,X).

```
1. Read the first tuple from the stream and store it as the state tuple, denoted
   by $x_s$.
2. Read the next X tuple ($x_b$) and do :
       if "$x_s$.TS=$x_b$.TS", replace $x_s$ with $x_b$ as the state tuple
       else (i.e., "$x_s$.TS<$x_b$.TS" holds)
           if "$x_s$.TE$\leq x_b$.TE", replace $x_s$ with $x_b$ as the state tuple
           else (i.e., $x_b$'s lifespan is contained within that of $x_s$)
               $x_b$ is output and $x_s$ remains as the state tuple.
3. Repeat Step 2 until all tuples have been read.
```

It is interesting to consider using a semijoin algorithm as a preprocessor for a join operation. Intuitively, the advantages are: (1) the output stream from a semijoin operation has the same sort ordering as the input stream — *order-preserving*; (2) with proper sort orderings, the semijoin algorithms scan input streams only once and eliminate a number of "dangling" tuples, and thus the size of workspace for subsequent join operations may be reduced.

4.2.2.3 Intersect and Before Operators

In this section, we briefly consider the intersect-join and before-join operators. The effect of various sort orders on the intersect-join operator are listed in Table 4.3, which shows that the only cases in which stream processing is efficient

sort orders				intersect-join(X,Y)	intersect-semijoin(X,Y)
relation X		relation Y			
TS	$\uparrow$	TS	$\uparrow$	(a)	(b)

(*) Other sort orderings are not appropriate and therefore they are not listed here.

(a) state = {X tuples whose lifespan span y_b.TS} $\cup$
{Y tuples whose lifespan span x_b.TS} $\cup$
{Y tuples whose TS value lie in l} if $y_b.TS > x_b.TS$
{X tuples whose TS value lie in l} if $x_b.TS > y_b.TS$.

(b) local workspace = <Buffer-x, Buffer-y>.

Table 4.3: Effect of various sort orders on the intersect-join and intersect-semijoin

are when (1) both operands are sorted on TS in ascending order, or (2) both operands are sorted on TE in descending order.

We mention earlier that the best approach for implementing before-join appears to be the nested-loop join. It is easy to verify that there is no sort ordering that would significantly limit the amount of state information required when the before-join is implemented by a stream processor. However, we do not mean to imply that sorting is useless for nested-loop joins; with proper sort orders, nested-loop join can avoid scanning the inner relation in its entirety. For before-semijoin, one can easily devise a simple algorithm which scans both operand relations only once and is independent of any sort orderings. For brevity, we omit the detail.

4.3 Semantic Query Optimization

Semantic query optimization techniques have been introduced and shown to be potentially useful in many studies [Kin81, Chak84, Jar84, She89]. However the techniques have not been widely used in conventional systems. The reason, we speculate, might be that conventional application domains are seldom rich enough in semantics, i.e., they contain only a few useful semantic constraints which the query optimizer can profitably exploit. For temporal databases, time is unarguably rich in semantics and many temporal semantic properties/constraints do occur naturally. It is therefore our belief that, unlike conventional applications, semantic query optimization can play a significant role in temporal databases. In this section, we discuss informally the significance of semantic query optimization in temporal query processing.

Earlier we mention an interesting integrity constraint in the Faculty relation, namely the chronological ordering of data values which the attribute Rank can assume — assistant, associate and full. For every faculty member, being an assistant professor must occur before being promoted to an associate professor, which must then occur before becoming a full professor.

There are two consequences if the database system does not capture and use this constraint. First, and most important, the optimizer would not be able to recognize that the inequality join in the Superstar example is in fact a contained-semijoin. The inequality join operation shown in Figure 4.1(b) can be described pictorially using Figure 4.7(a). The equi-join on "f1.Name=f2.Name" shown in Figure 4.1(b) concatenates those f1 and f2 tuples corresponding to those assistant professors promoted to full professors. The inequality join then selects those f1 and f2 tuple pairs which satisfy the less-than join condition (θ') as shown in Figure 4.7(a). With the above semantic constraint, it is not difficult to see that

"f1.TS<f3.TE and f3.TS<f2.TE"

are redundant and the less-than join condition can be reduced to a contained-semijoin condition as shown in Figure 4.7(b). Being able to recognize a contained-semijoin allows the database system to make use of sort orderings and therefore the stream processing techniques discussed in the previous section.

Taking this example one step further, suppose that there is no re-hiring of faculty members, e.g., no assistant professors left the university and then later were re-hired as full professors. That is, in Figure 3.1 "$te_1=ts_2$" and "$te_2=ts_3$" are always true. In addition, suppose that all faculty members are hired as assistant professors. With this continuous employment assumption, the Superstar query can be transformed into: *List associate professor X if there exists another associate professor Y such that X is promoted from assistant professor level later than Y, but X is promoted to full professor rank earlier than Y.* The relational algebraic expression for this query can be simplified into:

$$\pi_{\mathrm{i.Name,i.TS,i.TE}}(\text{contained-semijoin}(\sigma_{\theta''}(\text{Faculty}_i), \sigma_{\theta''}(\text{Faculty}_j)))$$

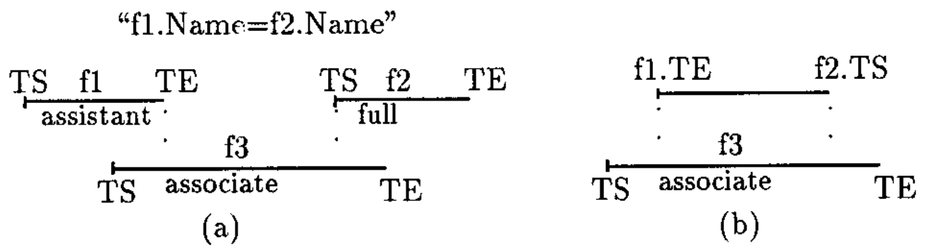


Figure 4.7: (a) The inequality join in the Superstar query, and (b) its equivalent contained-semijoin condition after semantic optimization

where $\theta'' \equiv$ "Rank=associate". As shown in Figure 4.7(b), the period [f1.TE, f2.TS) is actually the time during which the faculty member is at the associate professor level. When the associate professor tuples are sorted on the TS timestamp in ascending order (or we explicitly sort on this attribute), the algorithm contained-semijoin(X,X) discussed in the last section can be used to perform the semijoin which requires only a single scan of tuples (i.e., all associate professor tuples) and the local workspace is composed of only a state tuple and an input buffer. For this particular query, the stream processing algorithm can be extremely efficient.

The second consequence of the constraint on the Rank attribute is that we are able to eliminate two redundant inequalities in θ'; their presence makes it harder to recognize the join as contained-semijoin and there is also some overhead due to testing redundant qualification. Eliminating redundant qualifications is indeed a by-product of semantic query optimization.

4.4 Conclusions

We illustrate the deficiencies of conventional systems for temporal query processing using the complex Superstar query. This example leads to several observations which suggest new requirements for temporal query processing strategies. The most interesting and important observation is that inequality joins occur more often and naturally in temporal queries, and often contain a conjunction of a number of inequalities. For the Superstar example, it may be more efficient to implement the inequality join using contain-semijoin instead of using nested-loop join algorithm especially when tuples are properly sorted. These observations motivate our investigation of the stream processing strategies and suggesting new avenues of research in temporal query optimization techniques.

We propose stream processing techniques for processing various temporal join and semijoin operators. Given data integrity constraints and a temporal query, we discuss the effect of various sort orderings of streams of tuples on the efficiency with which the operator is implemented and the local workspace requirement in the stream processing environment. In particular, we note that the optimal sorting order may depend on the query itself and the statistics of data instances. We also briefly discuss semantic query optimization in temporal databases. As we mention in the previous chapter, temporal relations are augmented with timestamps such as TS and TE. However, database users are not allowed to update timestamps directly although a set of temporal operators are provided for data manipulation. From an algebraic manipulation point of view, these system-defined timestamps are the same as any user-defined integer attributes. The main difference becomes evident when the semantics of TS and TE timestamps are utilized in the semantic query optimization process. As we can see from the Superstar example, the system might not be able to evaluate the query using contained-semijoin without knowing the "intra-tuple" integrity constraint.

CHAPTER 5

Generalized Data Stream Indexing

In the previous chapters, we note that temporal joins (such as intersect-join) often contain a conjunction of several inequality predicates, and argue that conventional join methods such as nested-loop join and hash-join are inefficient or inappropriate for this type of join operations. In this chapter, we study the processing of the *snapshot* or *interval* queries. That is, the query refers to tuples that are active as of a particular time or over a certain period of time interval in the past as opposed to all tuples in the entire relation lifespan. We propose an indexing strategy that is appropriate for a certain subclass of complex temporal inequality join queries that are qualified with snapshot operators such as the as of and intersect operators. The strategy is to provide an indexing mechanism such that tuples in proximity of the query-specific time interval or time point can be retrieved efficiently. We discuss the advantages and limitations of the proposed scheme from the query language point of view. A quantitative analysis of the proposed scheme in terms of storage cost is presented, and optimization alternatives for reducing the storage requirement are proposed. We compare the proposed scheme with conventional indices (such as B^+tree) and discuss under what circumstances the proposed scheme is more efficient.

The organization of this chapter is as follows. In Section 5.1 we discuss the notion of checkpointing a temporal query execution. Section 5.2 is devoted to the query processing algorithms using checkpoints and their indices. A quantitative analysis is presented in Section 5.3, and some optimization alternatives are proposed in Section 5.4. Section 5.5 contains a discussion of related work and a brief summary.

5.1 Checkpointing Query Execution

We discuss the notion of *checkpointing* the execution state of a query (along the time dimension) in the context of stream processing presented in the previous chapter, and the indexing of checkpoints based on the checkpoint times.

5.1.1 Background Information

We consider the following subclass of queries for specifying the generalized data stream indices:

- TSJ_1 queries where the query qualification do not contain any comparison predicates involving timestamps, and
- all select queries that do not involve comparison with timestamps. For example, given a relation X(S,U,TS,TE), the predicate "U>10" is an acceptable query qualification of a select query that we consider here.

The reason for focusing only on queries that do not involve comparison predicates with timestamps is that these queries can be used to specify what information will be stored in the index. Allowing comparison predicates with timestamps will limit the use of data stream indices to a specific "time window" and thus the indices may be of little use. This becomes clear when we discuss the proposed technique below.

Recall that we consider data streams which are time-interval temporal relations sorted on a timestamp (i.e., either TS or TE). For simplicity of explanation, we consider only data streams that are sorted in increasing order. That is, tuples in a data stream can be efficiently accessed one at a time and in the order of successive timestamp values using the data stream pointer.

5.1.2 The New Approach

To illustrate the idea more clearly, we consider a stream processor that implements a query $Q \in TSJ_1$ with data streams X and Y as shown in Figure 5.1. A stream processor that implements the processing of the query Q starts by reading elements at the beginning of data streams. At any time point t, the execution state of the stream processor includes:

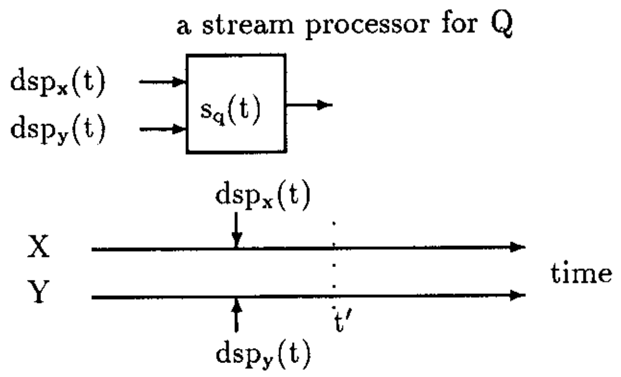


Figure 5.1: Checkpointing the execution of a stream processor for query Q

- state information, denoted as $s_q(t)$, stored in the local workspace of the stream processor.
- $dsp_x(t)$ and $dsp_y(t)$: the data stream pointers for X and Y respectively which represent the position of the data stream at which the stream processor has read so far. Recall that data stream elements are accessed one at a time using the data stream pointer.

A checkpoint of the execution state of the query Q has the following characteristics in terms of the state information and data stream pointers:

- The execution state at time t is stored in a checkpoint, denoted as $ck_q(t)$.
- Consider a time point t' which is greater than t. The execution state at t' is a function of $ck_q(t)$ and all tuples in the data stream X (respectively Y) between $dsp_x(t)$ (respectively $dsp_y(t)$) and the first tuple in the data streams after t'. That is, the execution state at t' contains sufficient information so that re-reading the portions of data streams prior to $dsp_x(t)$ and $dsp_y(t)$ can be avoided.

The state information required depends on the query itself and we will define the state information shortly. Intuitively, at any time the state information of a join query consists of a subset of tuples of operand relations that are previously read.

In our approach the data stream indices are built by:

- periodically checkpointing the execution of Q on data streams X and Y along the time dimension, and
- checkpoints are in turn indexed on their checkpoint times as depicted in Figure 5.2.

In the next section, we will define checkpoints formally, but to put it simply, a checkpoint (e.g., $ck_q(t_2)$ in Figure 5.2) at a time point (i.e., t_2) contains some information about the execution of Q on X and Y such that the response of an interval query (e.g., Q **intersect** $[t_2^+, t)$ where $t_2 < t_2^+ < t_3$) can be obtained in the following way:

> Find the appropriate checkpoint (e.g., in this case $ck_q(t_2)$) using the time index on checkpoints, then access tuples in the operand data streams appended after t_2. "Continue" the execution of the query (e.g., in this case Q) using the tuples thus accessed until t.

Since not all tuples of the operand data streams can be randomly accessed, one can regard this approach as creating a sparse index on data streams using Q. The sequence of checkpoints and the time index of checkpoints form the foundation of the generalized data stream index proposed in this chapter. For convenience, we refer to the query Q as the *indexing condition*[1].

5.1.2.1 Creating Checkpoints

We first define the term *state predicate*, and then discuss its role in the generalized data stream indexing technique and how checkpointing is performed.

Roughly speaking, a checkpoint can be viewed as a snapshot of the current database content as of the checkpoint time. That is we store the tuple identifiers (TID's) of tuples that are active as of the checkpoint time as checkpoints, and the temporal relations are periodically checkpointed. However, in general storing the snapshot database content is extremely expensive. To limit the tuples that we are interested in and are stored in the checkpoints, we allow using a *state predicate* for every relation, which is defined as follows.

[1] More generally, the indexing condition Q can be a query that subsumes a set of frequently asked queries. For example, Q can be $\sigma_{\text{intersect-join}(X,Y)}(X,Y)$.

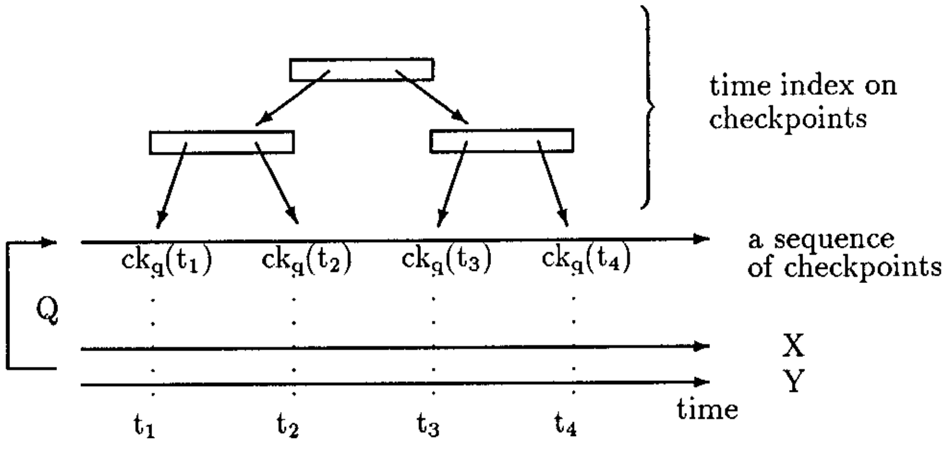

time index on checkpoints
a sequence of checkpoints
$ck_q(t_1)$ $ck_q(t_2)$ $ck_q(t_3)$ $ck_q(t_4)$
Q
X
Y
time
t_1 t_2 t_3 t_4

Figure 5.2: Time index on checkpoints

Definition 5.1 A *state predicate* for a relation R, denoted as $P|_r$, is a query qualification on R. That is, $P|_r$ is a conjunction of several comparison predicate. □

In other words, only those tuples which satisfy the corresponding state predicate will be part of the checkpoints. The flexibility of specifying which tuples are of most concern allows us to build a *generalized* data stream index.

Given an indexing condition $Q \equiv \sigma_{P(X,Y)}(X,Y) \in TSJ_1$, we can derive state predicates for both data streams X and Y. A state predicate for data stream X, denoted as $P|_x$, is obtained from P(X,Y) by substituting join predicates and comparison predicates that involve the data stream Y with "true"[2]. That is, $P|_x$ contains only comparison predicates involving only X in P(X,Y). The state predicate for the relation Y denoted as $P|_y$ is defined analogously.

Example 5.1 Consider the film industry examples presented earlier. The query to find the head of a studio that the director "Fred" worked for at the same time is:

$$\sigma_{\text{intersect-join(Studio,Dir)} \wedge \text{Studio.Sname=Dir.Sname} \wedge \text{Dir.Dname=Fred}}(\text{Studio,Dir}).$$

[2] Recall that we consider only conjunctions of join and comparison predicates as query qualification.

The state predicate for the relation Dir is "Dname=Fred" while the state predicate for the relation Studio is "true". □

The above "top-down" approach derives state predicates using the given indexing condition. Alternatively, users can provide the state predicates for individual data stream, i.e., a "bottom-up" approach. In this way, users specify a state predicate for each data stream. For example, suppose the state predicates for data streams X and Y are $\mathrm{P}|_{\mathrm{x}}$ and $\mathrm{P}|_{\mathrm{y}}$ respectively. The indexing condition, which is a more general query (by default), is:

$$\mathrm{Q} \equiv \sigma_{\mathrm{P}|_{\mathrm{x}} \wedge \mathrm{P}|_{\mathrm{y}} \wedge \text{intersect-join}(\mathrm{X},\mathrm{Y})}(\mathrm{X},\mathrm{Y}).$$

Given the state predicates $\mathrm{P}|_{\mathrm{x}}$ and $\mathrm{P}|_{\mathrm{y}}$, we can now define a checkpoint. Three kinds of information are stored in a checkpoint:

- the *checkpoint time*,
- the *state information*, and
- the *data stream pointers.*

For a checkpoint $\mathrm{ck_q(t)}$ performed at time point t, let the checkpoint prior to $\mathrm{ck_q(t)}$ be denoted as $\mathrm{ck_q(t^-)}$ performed at time $\mathrm{t^-}$, and the next checkpoint be performed at time $\mathrm{t^+}$. The checkpoint $\mathrm{ck_q(t)}$ contains[3]:

1. The checkpoint time is t.
2. State information: there are two cases depending on whether the data stream is sorted on TS or TE timestamp:

   TS — We first consider when the data stream X is sorted on the TS timestamp. The state information of the data stream X, denoted as $\mathrm{s_x(t)}$, contains the tuple identifiers (TID's) of all tuples $x \in \mathrm{X}$ such that "$x.\mathrm{TS}<\mathrm{t} \wedge \mathrm{t} \leq x.\mathrm{TE} \wedge \mathrm{P}|_{\mathrm{x}}$" holds, where $\mathrm{P}|_{\mathrm{x}}$ is the state predicate for the data stream X. Basically the state information contains tuples

[3] That is, $\mathrm{t^-} < \mathrm{t} < \mathrm{t^+}$. If there is no such $\mathrm{ck_q(t^-)}$, $\mathrm{ck_q(t^-)}$ and $\mathrm{t^-}$ are assumed to be an empty set and 0 respectively.

which are active as of the checkpoint time and satisfy the state predicate. Note that tuples in $s_x(t)$ either belong to $s_x(t^-)$ or start during the interval $[t^-,t)$.

TE — We now consider when the data stream X is sorted on the TE timestamp. The state information of the data stream X contains the tuple identifiers (TID's) of all tuples $x \in X$ such that "$x.TS<t^+ \wedge t^+ \leq x.TE \wedge P|_x$" holds, where $P|_x$ is the state predicate for the data stream X. Basically the state information contains tuples which are active during the interval $[t,t^+)$ and satisfy the state predicate. Note that the state information at checkpoint time t depends on the next checkpoint time t^+.

The state information for data stream Y can be obtained analogously.

3. Data stream pointer: there are two cases depending on whether the data stream is sorted on TS or TE timestamp:

   TS — The data stream pointer for X contains the TID of tuple $x \in X$ such that x has the smallest TS value in X but greater than or equal to t.

   TE — The data stream pointer for X contains the TID of tuple $x \in X$ such that x has the smallest TE value in Y but greater than or equal to t.

   Using the data stream pointer, one can access the first tuple that is appended in the data stream after the checkpoint time t (and subsequent tuples as well). The data stream pointer for data stream Y can be obtained analogously.

In addition to the three basic types of checkpoint information for each operand data streams X and Y, one can also store the TID's of matching tuple pairs as the *incremental result* in checkpoints. Given a checkpoint $ck_q(t)$ at time t and its next checkpoint time t^+, we denote X_t and Y_t as the portion of data streams X and Y respectively that are appended during $[t,t^+)$. Note that X_t and Y_t can be accessed via the data stream pointers $dsp_x(t)$ and $dsp_y(t)$ respectively. We also denote $S_x(t)$ and $S_y(t)$ as the tuples retrieved using the TID's in the state information $s_x(t)$ and $s_y(t)$ respectively.

The incremental result of Q stored at t, denoted as $ir_q(t)$, contains the TID's of the following matching tuple pairs:

checkpoints	$ck_{q_1}(t_0)$	$ck_{q_1}(t_1)$	$ck_{q_1}(t_2)$
t	t_0	t_1	t_2
$s_x(t)$	$\{x_0, x_1\}$	$\{x_1, x_3\}$	$\{x_5, x_6\}$
$dsp_x(t)$	$\{x_2\}$	$\{x_4\}$	$\{x_7\}$

Table 5.1: Checkpoints of Q_1 in Figure 5.3

$$\sigma_P(\ (X_t \cup S_x(t)),\ (Y_t \cup S_y(t))\ )$$

that is, the tuple pairs that satisfy the join condition. A pair of TID's need not be stored if both TID's have been stored in the state information at checkpoints. Generally speaking, storing incremental result of Q may require a significantly large amount of space and thus it is likely very expensive. When we discuss the query processing algorithms in the next section, we will focus on the three basic types of checkpoint information which can always be used to compute the incremental result.

We illustrate the approach using several examples. We start with Example 5.2 in which the indexing condition is a simple select query on a data stream X.

Example 5.2 Suppose the data stream X(S,U,TS,TE) is sorted on TS and consider $Q_1 \equiv \sigma_{U>10}(X)$ as the indexing condition. That is, the state predicate $P|_x$ is "U>10". The checkpoint ck at time point t contains:

1. The checkpoint time is t and $dsp_x(t)$ contains the TID of tuple $x \in X$ as defined earlier.

2. The state information at checkpoint time t, $s_x(t)$, contains tuples $x \in X$ such that "$x.TS<t \wedge t \leq x.TE \wedge x.U>10$" holds.

Consider an example data stream in Figure 5.3, in which we assume that all tuples satisfy the state predicate "U>10". Assuming there is no checkpoint prior to $ck_{q_1}(t_0)$, the contents of checkpoints $ck_{q_1}(t_0)$, $ck_{q_1}(t_1)$ and $ck_{q_1}(t_2)$ are listed in Table 5.1. □

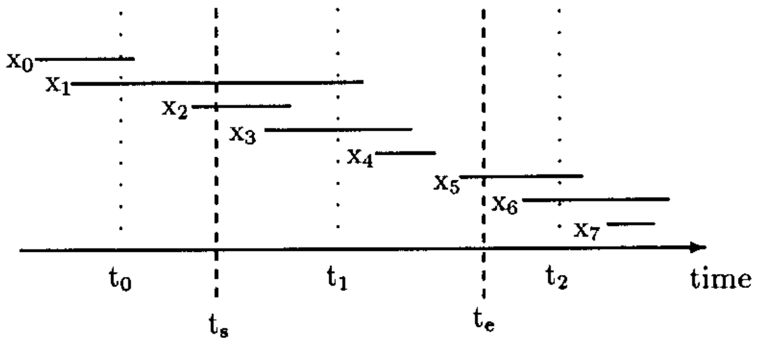


Figure 5.3: Data stream sorted on TS: checkpointing the query Q_1

Example 5.3 Consider that both data streams X and Y are sorted on the TS timestamp as shown in Figure 5.4, and a query $Q_2 \equiv \sigma_{\text{intersect-join(X,Y)}}(X,Y)$ as the indexing condition:

1. The checkpoint time is t, and the data stream pointers contains the TID's of tuples from X and Y as defined earlier.

2. The state information at checkpoint time t contains tuple $x \in X$ and tuple $y \in Y$ that are active at t. Note that the state predicates ($P|_x$ and $P|_y$) are "true" in this example.

In Table 5.2 we list the three types of information in the checkpoints as well as the incremental result that can be stored in checkpoints. □

Example 5.4 This example differs from the previous one in that both data streams X and Y are sorted on the TE timestamp as shown in Figure 5.5:

1. The checkpoint time is t, and the data stream pointers contains the TID's of tuples from X and Y as defined earlier.

2. Suppose the next checkpoint time is determined to be t^+, the state information at checkpoint time t contains tuple $x \in X$ and tuple $y \in Y$ that are active at t^+. As in the previous example, the state predicates ($P|_x$ and $P|_y$) are "true".

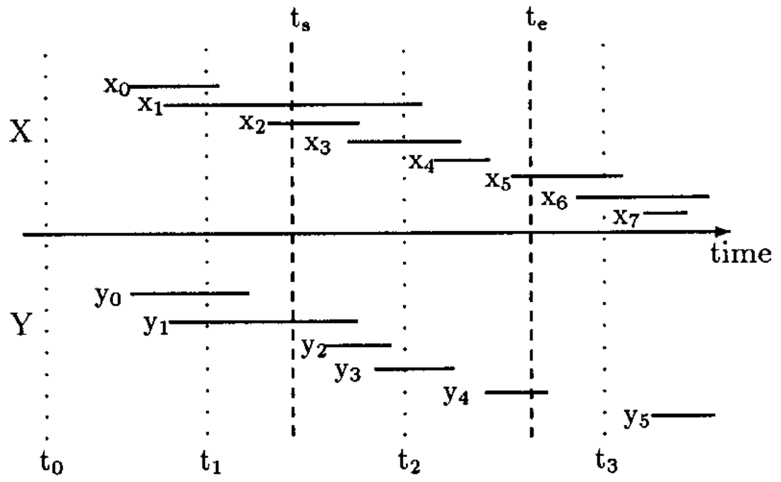


Figure 5.4: Data streams sorted on TS: checkpointing the query Q_2

checkpoints	$ck_{q_2}(t_0)$	$ck_{q_2}(t_1)$	$ck_{q_2}(t_2)$	$ck_{q_2}(t_3)$
t	t_0	t_1	t_2	t_3
$s_x(t)$	{ }	$\{x_0,\ x_1\}$	$\{x_1,\ x_3\}$	$\{x_5,\ x_6\}$
$s_y(t)$	{ }	$\{y_0,\ y_1\}$	$\{y_3\}$	{ }
$dsp_x(t)$	$\{x_0\}$	$\{x_2\}$	$\{x_4\}$	$\{x_7\}$
$dsp_y(t)$	$\{y_0\}$	$\{y_2\}$	$\{y_4\}$	$\{y_5\}$
$ir_{q_2}(t)$	{ }	$\{ <x_1,y_2>, <x_2,y_1>, <x_2,y_2>, <x_3,y_1>, <x_3,y_2> \}$	$\{ <x_4,y_3>, <x_4,y_4>, <x_5,y_4> \}$	{ }

Table 5.2: Data streams sorted on TS: checkpoints of Q_2 in Figure 5.4

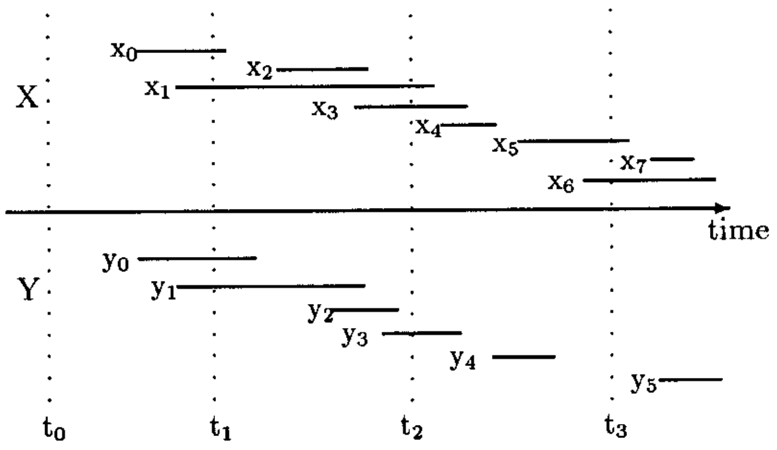


Figure 5.5: Data streams sorted on TE: checkpointing the query Q_2

In Table 5.3 we list the three types of information in the checkpoints as well as the incremental result that can be stored in checkpoints. □

Let us now discuss the situations when tuples are appended to the data stream. Recall that we adopt the append-only update policy. We first consider the data streams sorted on the TS timestamp, i.e., current tuples are also in the data streams. When a current tuple (i.e., <s,u,t_s,*now*>) is updated (i.e., *now* is set to a specific value), its TID may have to be stored at multiple checkpoints as state information if the tuple satisfies the corresponding state predicate[4]. For data streams sorted on the TE timestamps, the corresponding TID may have to be stored at multiple checkpoints when a history tuple (i.e., <s,u,t_s,t_e>) is appended to the data stream.

5.1.2.2 Time Index on Checkpoints

Given a sequence of checkpoints as illustrated in Figure 5.2, one can easily build a time index on checkpoints based on the checkpoint times. That is, given a time point t, the checkpoint taken at t, or the previous checkpoint or the next checkpoint can be accessed directly. Moreover, conventional methods such as B$^+$tree

[4] From a different perspective, we are allowing late updates and therefore the state information at some checkpoints may have to be refreshed accordingly.

checkpoints	$ck_{q_2}(t_0)$	$ck_{q_2}(t_1)$	$ck_{q_2}(t_2)$	$ck_{q_2}(t_3)$
t	t_0	t_1	t_2	t_3
$s_x(t)$	$\{x_0, x_1\}$	$\{x_1, x_3\}$	$\{x_5, x_6\}$	$\{\}$
$s_y(t)$	$\{y_0, y_1\}$	$\{y_3\}$	$\{\}$	$\{\}$
$dsp_x(t)$	$\{x_0\}$	$\{x_0\}$	$\{x_1\}$	$\{x_5\}$
$dsp_y(t)$	$\{y_0\}$	$\{y_0\}$	$\{y_3\}$	$\{y_5\}$
$ir_{q_2}(t)$	$\{\}$	$\{ <x_1,y_2>, <x_2,y_1>, <x_2,y_2>, <x_3,y_1>, <x_3,y_2> \}$	$\{ <x_4,y_3>, <x_4,y_4>, <x_5,y_4> \}$	$\{\}$

Table 5.3: Data streams sorted on TE: checkpoints of Q_2 in Figure 5.5

can be used for implementing this type of indexing. For example, checkpoints are stored at leaf nodes of a B^+tree as variable length records.

5.2 Query Processing using Data Stream Index

In this section, we discuss the processing algorithms for some types of complex temporal snapshot or interval queries using the proposed checkpointing and indexing scheme, and discuss their limitations.

Suppose we have a generalized data stream index based on the indexing condition $Q \equiv \sigma_P(X,Y) \in TSJ_1$. Let $\sigma_{P'}(X,Y) \in TSJ_1$ and Q' be a query of the following form:

$Q' \equiv \sigma_{P'}(X,Y)$ **intersect** $[t_s,t_e)$, or

$Q' \equiv \sigma_{P'}(X,Y)$ **as of** t_s, where $t_s \neq now$.

In order to use the data stream index for processing Q', one has to obtain two predicates from P' as in the case for state predicates — $P'|_x$ and $P'|_y$. That is, $P'|_x$ (respectively $P'|_y$) is obtained by replacing all terms in P' that involve Y (respectively X) with "true"[5]. Furthermore, we require that $P'|_x \Rightarrow P|_x$ which is the state predicate that is used to determine and store the state information of data stream X. Similarly, we require that $P'|_y \Rightarrow P|_y$. The implications are necessary because the state information in checkpoints obtained using P has to be a superset of the state information that would have been obtained using P′ instead of P, and therefore the checkpoints contain sufficient information for query processing. The query processing algorithm that uses the data stream index for the **intersect** queries is stated as follows.

Algorithm 5.1 Using the State Information and Data Stream Pointers.

1. Given the query specific interval $[t_s, t_e)$, access the latest checkpoint, denoted as ck_s, prior to t_s using the time index on checkpoints. Let the checkpoint time of ck_s be t, and the data stream pointers be $dsp_x(t)$ and $dsp_y(t)$. If a data stream is sorted on the TE timestamp, the latest checkpoint prior to t_e, denoted as ck_e, is also accessed.

2. Retrieve the tuples using TID's in the state information that are stored in the checkpoints ck_s (or in ck_e if the data stream is sorted on TE), and apply the predicates $P'|_x$ and $P'|_y$ on tuples in X and Y respectively.

3. Retrieve tuples in X and Y which are appended during $[t, t_e)$ by following the data stream pointers $dsp_x(t)$ and $dsp_y(t)$, and apply the predicates $P'|_x$ and $P'|_y$ respectively.

4. The set of all tuples from steps (2) and (3) contains all the tuples that should participate in the join. Select tuple pairs that satisfy the user query qualification P′. Note that the tuples that have to be kept in the workspace are limited to tuples spanning a common point in time. □

For the **as of** queries, the query processing algorithm remains essentially the same except:

In the step 3, tuples that are appended during the interval of $[t, t_s]$ (instead of $[t, t_e)$) are accessed.

[5] More restrictive predicates ($P'|_x$ and $P'|_y$) may be obtained by using constraint propagation algorithms [Ull82, Chak84, Jar84] which will also be described in a later chapter.

With the data stream indices, it can be shown that the following classes of queries can also be processed using the above algorithm:

1. $\sigma_{P'}(X)$ **intersect** $[t_s,t_e)$, where $P' \Rightarrow P|_x$.
2. $\sigma_{P'}(Y)$ **intersect** $[t_s,t_e)$, where $P' \Rightarrow P|_y$.

Example 5.5 Consider the query $\sigma_{\text{intersect-join(X,Y)}}(X,Y)$ **intersect** $[t_s,t_e)$ in Example 5.3 where:

- the indexing condition is $\sigma_{\text{intersect-join(X,Y)}}(X,Y)$,
- both data streams X and Y are sorted on TS, and
- the checkpoints of the corresponding data stream index are shown in Figure 5.4 and Table 5.2).

In step (2), we retrieve tuples $\{x_0, x_1\}$ and $\{y_0, y_1\}$. By following the data stream pointers ($\{x_2, y_2\}$), the join operation in step (4) produces tuple pairs: $\{ <x_1,y_1>, <x_1,y_2>, <x_1,y_3>, <x_2,y_1>, <x_2,y_2>, <x_3,y_1>, <x_3,y_2>, <x_3,y_3>, <x_4,y_3>, <x_4,y_4>, <x_5,y_4> \}$. Note that had the incremental results been stored in checkpoints, this query can also be processed by using both the state information and incremental results (i.e., without using data stream pointers). □

Let us consider another processing strategy in which only the data stream pointers stored in checkpoints are used. Suppose that the data streams are sorted on TS, and we are interested in retrieving tuples that started during $[t_s, t_e)$. For example, consider:

- the indexing condition is $Q \equiv \sigma_P(X)$, and
- the user query is $Q' \equiv \sigma_{P' \wedge X.TS \textbf{ between } [t_s,t_e)}(X)$,
  where P' is a comparison predicate involving only non-time attributes (it is not required that P' implies P or vice versa).

For this type of queries, the query processing algorithm that uses the data stream pointer only is as follows.

Algorithm 5.2 Using the Data Stream Pointer only.

1. Access the latest checkpoint, denoted as ck_s, prior to t_s using the time index. Let the checkpoint time of ck_s be t.

2. Retrieve tuples which start in $[t,t_e)$ by following TID's in $dsp_x(t)$ and apply the query qualification P'. □

Example 5.6 Consider a query $\sigma_{V>10 \wedge X.TS \textbf{ between } [t_s,t_e)}(X)$ and the example data stream in Figure 5.3. The checkpoint prior to t_s is $ck_q(t_0)$. Following the $dsp_x(t_0)$, i.e., $\{x_2\}$, tuples x_2, x_3, x_4, x_5, and x_6 are retrieved in step (2). When the tuple x_6 is accessed, step (2) stops and x_6 is discarded from the response as its TS value is greater than t_e. The query response is $\{x_3, x_4, x_5\}$. □

Let us now discuss the limitations of the proposed checkpointing and indexing scheme. In the proposed scheme, only TSJ_1 queries and the select queries (both without comparison predicates involving timestamps) are allowed as the indexing conditions. Recall that for TSJ_1 join queries, the lifespans of all participating tuples have to intersect with each other. To understand the importance of this restriction, let us consider "before-join(X,Y)" whose join condition is "X.TE<Y.TS" as the indexing condition. That is, tuples that satisfy the join condition do not necessarily intersect. Given a tuple $x \in X$ which starts at some time t, we note that x may join with theoretically infinitely many "future" tuples $y \in Y$ which start after the tuple x ends. Or conversely, the tuple $y \in Y$ may join with theoretically infinitely many "past" tuples $x \in X$ which ends before the tuple y starts. For the query processing algorithms that are presented earlier to work properly, the TID of tuple x has to be stored at every checkpoint after the time point t. This requires significant storage space and renders the proposed scheme inefficient. With the restriction, we only need to store in a checkpoint the TID's of tuples that span the checkpoint time.

5.3 Quantitative Analysis

We consider the overhead of storing the state information in checkpoints. First, we list some required notation:

- λ denotes the mean rate of insertion of tuples into the relation.
- $\overline{\mathrm{T_{ls}}}$ denotes the average tuple lifespan.
- $\mathrm{TR_{ls}}$ denotes the relation lifespan.
- $\mathrm{size_{tuple}}$ denotes the tuple size in number of bytes.
- $\mathrm{size_{tid}}$ denotes the TID size in number of bytes.

Using Little's result [Lit61], the average number of active tuples of a relation at a random time, denoted as $\overline{\mathrm{n}}$, is given by:

$$\overline{\mathrm{n}} = \lambda \cdot \overline{\mathrm{T_{ls}}}$$

A reasonable assumption is that the number of active tuples at checkpoint times is also $\overline{\mathrm{n}}$. Similarly, the total number of tuples in the relation is:

$$\lambda \cdot \mathrm{TR_{ls}}$$

Suppose that the selectivity of the state predicate q for the state information of data stream X is σ_q, i.e., σ_q is the fraction of tuples in X that satisfy q. The number of TID's stored in the state information is:

$$\sigma_\mathrm{q} \cdot \mathrm{n_{ck}} \cdot \overline{\mathrm{n}} = \sigma_\mathrm{q} \cdot \mathrm{n_{ck}} \cdot \lambda \cdot \overline{\mathrm{T_{ls}}}$$

where $\mathrm{n_{ck}}$ is the number of checkpoints that have been taken. We define the overhead as the ratio of the storage size for state information over the relation size:

$$\sigma_\mathrm{q} \cdot \mathrm{n_{ck}} \cdot \overline{\mathrm{T_{ls}}} \cdot \mathrm{size_{tid}} / \{\mathrm{TR_{ls}} \cdot \mathrm{size_{tuple}}\}$$

This quantity is consistent with our intuition that the overhead is smaller for (1) relations with relatively short tuple lifespans (represented by the ratio $\overline{\mathrm{T_{ls}}}/\mathrm{TR_{ls}}$), and (2) more selective state predicate (i.e., σ_q is smaller).

5.4 Optimization: Reducing Storage Space

For the data stream indexing technique that is presented earlier, the state information may still require a large amount of storage space when many qualified tuples span the checkpoint times. The required storage space can be reduced as follows. Consider two time points t^- and t where $t^- < t$ as shown in Figure 5.6. Suppose that there are only a few active tuples at the time point t^- and there are a lot of insertions during the period between time points t^- and t. If we had chosen the time point t as the checkpoint time, many tuples may have to be included in the state information. From this point of view, we may prefer to choose t^- (whose tuple "density" is low) as the checkpoint time.

One can utilize this idea differently. Here we choose the time point t as the checkpoint time. Instead of storing in $dsp_x(t)$ the first tuple of X that is appended after t, we store the first tuple of X which is appended after t^-. As illustrated in Figure 5.6, $dsp_x(t)$ contains the TID of tuple x_2 instead of x_3. That is, we "reset" the data stream pointer such that it points "backward" by some distance to a point where the "density" of tuples is low. The state information $s_x(t)$ now contains the TID's of tuples which span the entire interval (t^-,t). Using the query processing algorithms presented earlier, processing queries would have to access more tuples, i.e., those which are appended during the interval (t^-,t). The tradeoff is that the state information $s_x(t)$ contains fewer tuple identifiers. Below we present a quantitative analysis of this storage reduction optimization alternative.

In the following we assume that all tuple lifespans are independent and exponentially distributed with mean $\overline{T_{ls}}$. Suppose that the distance between t^- and t is $\bar{t}$ for all checkpoint times t. We further suppose that the unit cost of reading a tuple using the data stream pointer (i.e., sequentially) is $cost_{seq}$ and that of reading a tuple using TID is $cost_{rand}$ which is generally more expensive. The expected value of the overhead of storing state information at all checkpoints becomes:

$$P[y>\bar{t}\,] \cdot \sigma_q \cdot n_{ck} \cdot \overline{T_{ls}} \cdot size_{tid}/\{TR_{ls} \cdot size_{tuple}\}$$

where y is a random variable representing the tuple lifespan and $P[y>\bar{t}\,]$ is the probability that a tuple would span the entire interval (t^-,t). The probability $P[y>\bar{t}\,]$ equals:

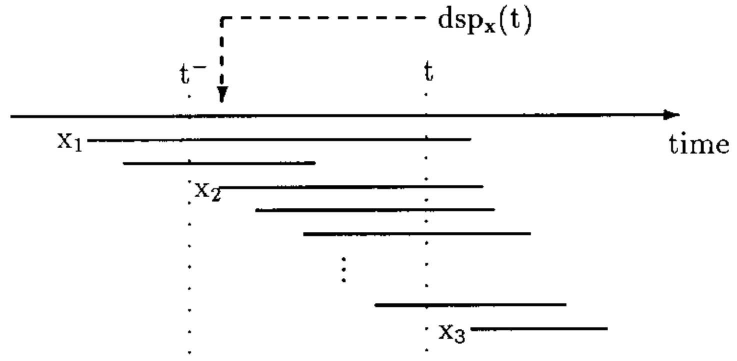


Figure 5.6: Optimization: reducing state information

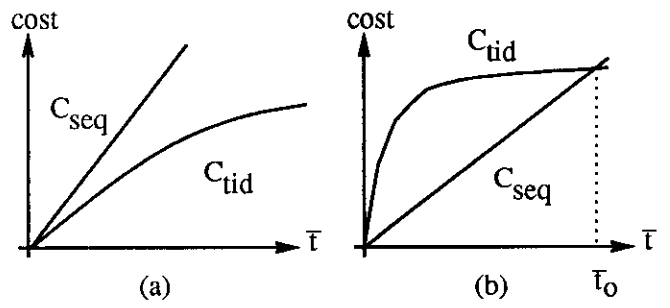


Figure 5.7: Tradeoffs in tuple retrieval times in the state information

$$1 - P[y \leq \bar{t}] = e^{-\bar{t}/\overline{T_{ls}}}$$

Note that the number of TID's (per checkpoint) that are eliminated from being stored in the state information is:

$$(1 - e^{-\bar{t}/\overline{T_{ls}}}) \cdot \sigma_q \cdot \lambda \cdot \overline{T_{ls}}$$

and the cost of reading the tuples via these TID's is:

$$C_{tid} = (1 - e^{-\bar{t}/\overline{T_{ls}}}) \cdot \sigma_q \cdot \lambda \cdot \overline{T_{ls}} \cdot cost_{rand}$$

The number of *extra* tuples that have to be read sequentially using the "reset" data stream pointer is $\lambda \cdot \bar{t}$ and the cost of reading these tuples is:

$$C_{seq} = \lambda \cdot \bar{t} \cdot cost_{seq}$$

The slopes of C_{tid} and C_{seq} at $\bar{t} = 0$ are:

$$\left.\frac{dC_{tid}}{d\bar{t}}\right|_{\bar{t}=0} = \sigma_q \cdot \lambda \cdot cost_{rand}$$
$$\left.\frac{dC_{seq}}{d\bar{t}}\right|_{\bar{t}=0} = \lambda \cdot cost_{seq}$$

Let C_r be the ratio of $(\sigma_q \cdot cost_{rand}/cost_{seq})$. For $C_r \leq 1$, there will be no intersection at any point in time other than time point 0 (Figure 5.7(a)). Let us now consider the cost difference:

$$D = C_{tid} - C_{seq}$$

An upper bound for the cross-over point (denoted as $\bar{t}_o$ in Figure 5.7(b)) is:

$$\bar{t}_{max} = \frac{\sigma_q \cdot cost_{rand}}{cost_{seq}} \cdot \overline{T_{ls}} = C_r \cdot \overline{T_{ls}}$$

The largest cost difference can be obtained by solving the following equation:

$$\frac{dD}{d\bar{t}} = \lambda \cdot (\sigma_q \cdot cost_{rand} \cdot e^{-\bar{t}/\overline{T_{ls}}} - cost_{seq}) = 0$$
$$\text{i.e., } \bar{t} = \log_e\left(\frac{\sigma_q \cdot cost_{rand}}{cost_{seq}}\right) \cdot \overline{T_{ls}} = \log_e(C_r) \cdot \overline{T_{ls}}$$

Solving the equation "D=0" numerically, the cross-over points ($\bar{t}_o$) for several C_r values are:

for $C_r = 1.5\ \overline{T_{ls}}$, $\bar{t}_o = 0.87\ \overline{T_{ls}}$
for $C_r = 2\ \overline{T_{ls}}$, $\bar{t}_o = 1.59\ \overline{T_{ls}}$

We note that as C_r exceeds $3\ \overline{T_{ls}}$, the cross-over point approaches $\bar{t}_{max}$.

5.5 Previous Work and Conclusions

Several temporal indices have recently proposed (e.g., [Rot87, Gun89, Kol89, Lom89, Elm90, Kol91]); they are extensions of traditional dense indexing methods such as B$^+$tree or multi-dimensional indices such as R-tree, and are based on *explicit* timestamp values in tuples. One can compare the storage requirement of these methods with the proposed scheme. For example, if we create a B$^+$tree index on the TS timestamp, there is an index entry in the B$^+$tree for every tuple in the relation. Recall that the relation lifespan is $\mathrm{TR_{ls}}$ and the rate of insertion of tuples is λ. The total number of TID's stored in the leaf nodes of the B$^+$tree, which is also the total number of tuples in the relation, is:

$$\mathrm{B} \;=\; \lambda \cdot \mathrm{TR_{ls}}$$

Assuming that the state predicate in our proposed scheme is "true" and thus the selectivity (σ_q) is 1. The number of TID's stored in the state information of all checkpoints is:

$$\mathrm{CK} \;=\; \mathrm{n_{ck}} \cdot \lambda \cdot \overline{\mathrm{T_{ls}}}$$

where $\mathrm{n_{ck}}$ is the number of checkpoints that have been taken[6]. The above two figures would be the same when:

$$\mathrm{n_{ck}} \;=\; \mathrm{TR_{ls}}/\overline{\mathrm{T_{ls}}}$$

Let us consider the difference in the number of TID's, i.e., D = B$-$CK. The derivative of D with respect to time is given by:

$$\frac{\mathrm{dD}}{\mathrm{dt}} \;=\; \lambda \cdot \frac{\mathrm{TR_{ls}}}{\mathrm{dt}} - \lambda \cdot \overline{\mathrm{T_{ls}}} \cdot \frac{\mathrm{dn_{ck}}}{\mathrm{dt}}$$

Note that the relation lifespan is continuously advancing and thus $\frac{\mathrm{TR_{ls}}}{\mathrm{dt}}$ is 1. Hence, the difference (D) remains unchanged if we checkpoint the query at a rate of:

$$\frac{\mathrm{dn_{ck}}}{\mathrm{dt}} \;=\; 1/\overline{\mathrm{T_{ls}}}$$

However, if the checkpointing frequency is smaller, the data stream index requires less storage space. Moreover, the checkpointing frequency should be smaller than $1/\overline{\mathrm{t}}_\mathrm{o}$ (where $\overline{\mathrm{t}}_\mathrm{o}$ is the cross-over point of the costs $\mathrm{C_{tid}}$ and $\mathrm{C_{opt}}$ in Figure 5.7(b))

[6] More precisely, each checkpoint also contains a data stream pointer. On the other hand, there are fewer non-leaf nodes in the time index on checkpoints compared with the B$^+$tree.

such that the storage reduction optimization strategy discussed earlier can be employed.

Other related work includes [Sto89, Val87]. In [Sto89], partial index has been proposed and is related to the notion of indexing condition presented in this chapter. Storing the TID's of joined tuple pairs as join indices is proposed in [Val87] and is similar to the idea of incremental results that can be stored in checkpoints.

To summarize, we propose a checkpointing and indexing scheme that is suitable in temporal database environment. The generalized data stream index can be used to efficiently process a subclass of complex joins qualified with a snapshot operator, especially when the query-specific time interval (e.g., in **as of** queries) is relatively short compared with the lifespans of data streams. This issue has not been addressed in other indexing approaches. Furthermore, it is envisioned that existing software (e.g., B^+tree) can be reused in the implementation of the indices. We also study the storage cost of this approach analytically and propose optimization techniques for reducing the storage requirement. Finally, we note that for large data streams which are less frequently accessed, a conventional index may need a large disk space and thus data stream indices become very attractive alternatives.

CHAPTER 6

Query Processing in Multiprocessor Database Machines

There are several classes of temporal queries. Among the most difficult to process is the multi-way joins (i.e., complex temporal pattern queries) whose join condition often contains a conjunction of several inequality join predicates. In general, these queries are often expensive to process. The difficulty of the problem can be further increased for the large temporal relations. Recently, there has been a growing interest in multiprocessor database machines which appear to have better price-performance than traditional DBMSs residing in mainframe computers. With the availability of relatively cheap parallel database machines, one should and can exploit parallelism for processing this type of queries — an approach that is seldom pursued. Moreover, a crucial design issue in these database machines is the *fragmentation strategy* which specifies how tables are fragmented and stored in the database system. The fragmentation strategy has a great impact on the efficiency of query processing. However, fragmentation strategies for temporal data have been largely ignored in temporal database environment.

In Chapter 4 we propose stream processing algorithms for processing temporal inequality join and semijoin operations. In Chapter 5 we propose the checkpointing and indexing scheme for processing temporal snapshot and interval join queries. In this chapter, we develop parallel strategies for TSJ_1 join queries based on the stream processing paradigm and the checkpointing and indexing scheme, and show the parallel strategies can be attractive alternatives. Recall that for a TSJ_1, all participating tuples that satisfy the join condition must share a common time point. For an inequality join of two relations, a straightforward approach is to dynamically and fully replicate the smaller operand relation among all processors. The parallel strategies proposed here are based on partitioning temporal relations on timestamp values. An analytical model is developed for estimating the number of tuples that have to be replicated; this model indicates in what situations only a fraction of a relation is replicated among processors as opposed to fully replicating the entire relation.

Another subclass of complex queries is the snapshot and interval join queries which are join queries as of a certain time point or over a certain time interval in the past. We also discuss optimizations can be achieved when these queries are processed using our proposed parallel strategies.

The organization of this chapter is as follows. Section 6.1 is devoted to a discussion of various existing fragmentation strategies for temporal relations. The parallel query processing strategies and optimization alternatives will be the main focus in Section 6.2. Implementation issues of the parallel strategies will be addressed in Section 6.3. Section 6.4 contains a discussion of other possible parallel query processing strategies. Finally, we discuss the related work and conclusions in Section 6.5.

6.1 Temporal Data Distribution

We consider a generic "shared-nothing" multiprocessor database machine [Sto86, DeW90] as illustrated in Figure 6.1, which has n nodes connected via an interconnection network. Each node has a processor, some main memory and secondary storage devices such as magnetic disks[1]. In this section, we discuss how a time-interval temporal relation can be partitioned and distributed in such a database machine, and the tradeoffs involved with respect to the efficiency of processing the various classes of queries: select, join and snapshot queries as listed in Chapter 3. Readers should keep in mind that whether or not a query qualification involves the partitioning attribute generally has a predominant impact on query processing efficiency, and therefore a specific fragmentation strategy can facilitate processing some kinds of queries while it may cause other queries to be more expensive to process.

A number of well-known fragmentation strategies have been proposed and implemented in multiprocessor database machines [Ter85, DeW90, Gha90]. They include:

[1] We use the terms nodes and processors interchangeably — a processor also refers to its processing capability and associated disks.

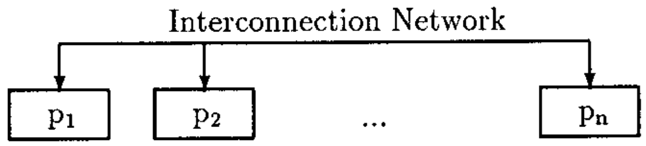


Figure 6.1: A generic "shared-nothing" multiprocessor database machine

- round-robin,
- hashing, and
- range-partitioning.

Hashing and Round-robin We first discuss the hashing and round-robin strategies as both strategies attempt to spread the workload more evenly among processors by distributing tuples *randomly*. For our discussion, we consider relations X(S,U,TS,TE) and Y(S,V,TS,TE) which can be fragmented on any attributes such as S, V or TS[2]. We first consider the case in which both relations are fragmented on their surrogates, i.e., S is the partitioning attribute.

Select queries with an equality predicate involving the partitioning attribute (such as "X.S=1") are processed by a unique processor. On the other hand, selection on non-partitioning attributes (e.g., "X.TS=50") and even range-searching on the partitioning attribute (e.g., "$50 \leq X.S \leq 100$") generally involves all processors. However, the parallel search is efficient only when the number of tuples processed on each processor is relatively large. The reason is that the overhead of starting and ending a subtransaction on a processor becomes small relative to the total work on a processor when the processor has to work on a large amount of tuples. If the select query has a highly selective predicate and there is an index on the search attribute, the parallel search is relatively inefficient as most processors would have wasted their resources on starting and committing a subtransaction which performs little work on searching tuples.

Let us now consider join queries. A naive approach to achieve parallelism is to fully and dynamically replicate the smaller relation at all processors, which

[2] The partitioning attribute can be a single attribute such as S or a "composite" attribute composed of multiple attributes such as S and TS.

is very expensive unless the relation is small. This expensive data movement does not occur when the join condition implies an equality predicate on the partitioning attribute (i.e., "X.S=Y.S") and both operand relations are fragmented using the same partitioning function (e.g., the same hash function). In general, data movement is required for inequality joins such as temporal join operations. For example, consider contain-join(X,Y) whose join condition is "X.TS<Y.TS ∧ Y.TE<X.TE". We note that tuples with close (but unequal) TS values are likely to be stored at different processors due to the randomness of hashing or round-robin strategies regardless of the partitioning attribute. Unless there is an additional qualification that limits the data that has to be examined, an expensive data movement is required prior to the execution of local joins, in parallel, on each processor.

A snapshot query generally requires all processors to participate since qualifying tuples are likely to be stored at all processors. As in the case of a select query, indexing techniques such as in [Elm90, Kol90] and the generalized data stream indices in Chapter 5 may be used to speed up the tuple retrieval locally.

Choosing the partitioning attribute involves several issues. First, the decision will depend on the frequency of various queries. For example, if most (both select and join) queries involve the surrogates, the surrogates may be good candidates for the partitioning attribute. Second, the decision may also depend on the attribute domain itself. As an example, suppose the domain of a time-varying attribute consists of only a small number of entities, say 50. For large relations, hashing on this attribute may produce a skewed data distribution and thus may have an adverse effect on query processing efficiency. Note that although surrogates are guaranteed to be unique in the database system (i.e., the values will not be re-used), a surrogate value may appear in a temporal relation more than once. To use the *uniqueness* of data values as the criterion of choosing the partitioning attribute, a composite attribute <S,TS> may be a better candidate[3].

Range-partitioning The range-partitioning strategy can be characterized by clustering and storing tuples with close (or equal) partitioning attribute values

[3] For continuous time-varying attributes (i.e., each object has one attribute value at any point in time), a value of the composite attribute <surrogate,timestamp> can uniquely determine a tuple. For this reason, one can also choose <S,TE> as the partitioning attribute.

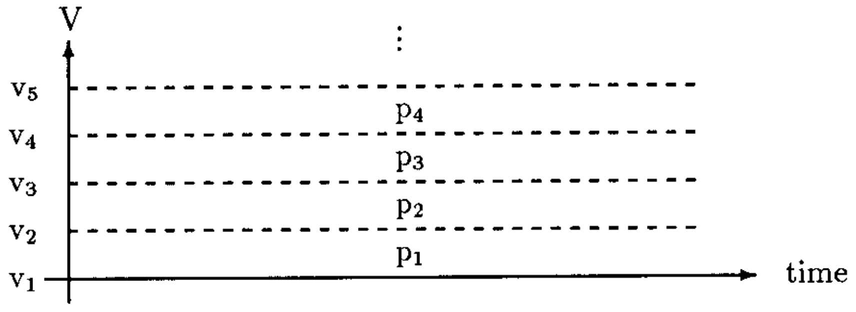


Figure 6.2: Range-partitioning along a non-time attribute

at the same processor. That is, the partitioning attribute is also the clustering attribute. We consider three types of range-partitioning strategies for a temporal relation Y(S,V,TS,TE).

Non-time Attribute The first strategy is to partition the relation Y along a non-time attribute dimension (e.g., V) as illustrated in Figure 6.2. This strategy partitions the attribute domain into a fixed number of intervals, and tuples in the same range are stored at the same processor. For example, as shown in Figure 6.2, tuples with V values in the range of $[v_1,v_2)$ are stored in processor p_1. The query processing algorithms for select, join and snapshot queries under this partitioning strategy are similar to the hashing and round-robin schemes that are discussed earlier. For example, select queries with an equality predicate involving the partitioning attribute (e.g., "V=1") are executed in a single processor. However, there are some (and perhaps slight) differences. First, range search queries on the partitioning attributes may sometimes be executed on only a subset of processors. For example, a range search "Y.V<v_3" is performed only at processors p_1 and p_2. For an equi-join involving the partitioning attribute (e.g., "X.U=Y.V"), the join can be executed in parallel without data movement if the join attributes are the partitioning attribute and both relations are fragmented using the same range-partitioning function.

In the simplest application of range partitioning, the number of partitions is the same as the number of processors in the system and therefore each processor is assigned a single partition. Recently it has been proposed in [Gha90] that

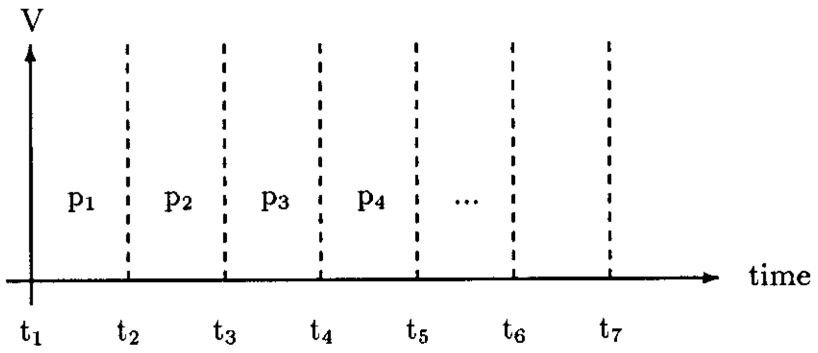


Figure 6.3: Range-partitioning along a timestamp

the domain is partitioned into a large number of smaller ranges and thus each processor is responsible for more than one partitions. This approach to processing select queries can be more efficient while keeping the workload among processors more balanced at the same time.

Time Attribute The second strategy is to partition the relation along a timestamp (e.g., TS) as illustrated in Figure 6.3. For example, tuples that start during the interval $[t_1,t_2)$ are stored in processor p_1. Using this scheme, equi-joins on non-partitioning attributes (e.g., non-time attributes) become more expensive to process as many tuples have to be moved among processors. However, partitioning relations based on a timestamp may be a good alternative for temporal query processing — below we discuss this scheme in more detail.

Let the processors be denoted as p_i, for $1 \leq i \leq n$. There are a total of n_{pi} intervals in the partitioning function:

$$[t_1,t_2), \quad \ldots, \quad [t_{n_{pi}-1},t_{n_{pi}}), \quad [t_{n_{pi}},t_{n_{pi}+1}).$$

We refer to t_i and $[t_i,t_{i+1})$ as *partitioning boundaries* and *partitioning intervals* (or simply *partitions*) respectively. Partitioning relations on the TS (respectively TE) timestamp is called TS (respectively TE) range-partitioning. As the relation lifespan is assumed to be $[0,now)$, by convention t_1 is 0 and $t_{n_{pi}+1}$ is *now*. In general, we require that the number of partitions be at least as large as the number of processors, i.e., $n_{pi} \geq n$. For simplicity, we adopt the hybrid range-partitioning scheme in [Gha90]: an interval $[t_j,t_{j+1})$ is assigned to p_i if i equals j modulo n. For TS range-partitioning, processor p_i stores a fragment of X, denoted as X_i, which contains tuples of X that start during the interval $[t_i,t_{i+1})$, i.e., "X.TS **between** $[t_i,t_{i+1})$" holds. Similarly for TE range-partitioning, X_i contains tuples that end during the partitioning interval.

Consider that both relations X and Y are TS range-partitioned using the same partitioning function. That is, tuples that start during the interval $[t_i,t_{i+1})$ are stored at processor p_i. For contain-join(X,Y), tuples with close TS values are likely to be clustered within the same processor and therefore a pair of tuples that satisfy the join condition are likely to be stored at the same processor. However, a partitioning boundary may lie between the start times of tuples x and y that satisfy the join condition and thus tuples x and y are actually stored at different processors. In short, processing temporal join in parallel may still require dynamically copying some fraction of tuples between the processors.

For range search queries and snapshot select queries involving the partitioning attributes, we note that the queries may sometimes be executed on only a subset of processors. For example, suppose we want to find the attribute values as of a certain time t_s which fails in the partition $[t_i,t_{i+1})$. As qualified tuples can start at any time earlier than t_s, processors from p_1 to p_i (which may be only a subset of processors) would generally have to participate in the search, unless there is some additional information which would limit the search to a smaller subset of processors.

In Chapter 4 we note that sorting relations on different timestamps may improve the efficiency of processing some queries in a centralized DBMS. For example, processing contain-semijoin(X,Y) using stream processing algorithms requires only minimal buffer space (enough for one tuple from each relation) when X is sorted on TS and Y is sorted on TE. Consider another example query: meet-join(Y,X) whose join condition is "Y.TE=X.TS". These queries may be more efficiently processed if the relation Y is range-partitioned on the TE timestamp and X is range-partitioned on the TS timestamp. In other words, selecting either TS or TE timestamp as the partitioning attribute may also depend on the types of common temporal join operations.

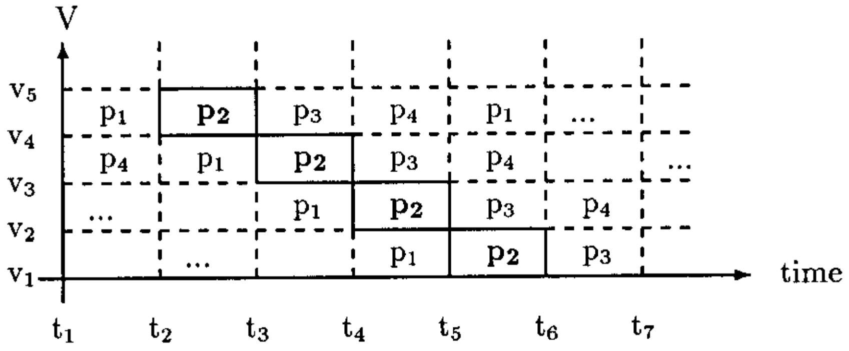


Figure 6.4: Two dimensional range-partitioning a temporal relation

2-dimensional Partitioning The above two range-partitioning schemes can be considered as one dimensional as there is only one partitioning attribute. A third strategy is to partition the relation in a two dimensional fashion; one can imagine we superimpose a Grid File structure [Nie84] on the search space. Each grid (i.e., rectangle) can be of different size and is assigned to a specific processor. One possible approach is illustrated in Figure 6.4: the search space is partitioned horizontally and vertically into regular bands, resulting in a number of rectangular grids of the same size. Grids within a vertical band are assigned to processors in a "staggering" fashion. For example, processor p_2 stores grids with solid boundaries as shown in Figure 6.4.

We note several interesting points with respect to query processing using multi-dimensional partitioning compared with the one dimensional schemes. For simplicity of explanation, let us compare the TS range-partitioning scheme and the two dimensional scheme (i.e., the TS timestamp is the partitioning attribute in both cases). For the two dimensional scheme, a range search may be executed on a subset of processors (as in the TS range-partitioning scheme). For example, tuples with condition "$Y.V>v_2 \wedge Y.V<v_3 \wedge Y.TS=t_4$" can be retrieved by processor p_2. Range search on only one partitioning attribute, however, involves some optimization issues. For example, in the TS range-partitioning scheme (i.e., one dimensional), a search on "$Y.TS=t_4$" is performed only at a single processor (i.e., p_4) whereas in the 2-dimensional scheme, the search is performed at all processors. That is, the two dimensional scheme attempts to spread the workload across all processors and thus reduce the query response time. Ideally, the

reduced query response time would be approximately about 1/n (where n is the number of processors) of the response time in the one dimensional scheme plus the overhead associated with coordinating the parallel search. This strategy may be beneficial if the number of tuples processed on each processor is relatively large. When the number of tuples processed is small (e.g., only a few tuples were inserted at time point t_4), the gain due to parallelism may not be offset by the associated overhead. For an equi-join that can be executed in parallel without data movement, the join condition has to imply "X.U=Y.V $\wedge$ X.TS=Y.TS" since there are two partitioning attributes. Note that the class of join queries to be processed in this fashion is smaller as the join condition is more restrictive. For this reason, join queries and snapshot join queries are generally more costly to process.

In the remainder of this chapter, we will further develop parallel query processing schemes based on range-partitioning on timestamp and show that these schemes may be good alternatives for complex temporal join queries and snapshot queries.

6.2 Parallel Temporal Query Processing

In this section, we discuss parallel processing strategies for complex temporal queries based on the following approach:

> Temporal relations are range-partitioned along the time dimension. Each processor will work *independently* on the partitions that are assigned to it. The query response is the union of results from all processors.

We first briefly revisit the notion of checkpointing the execution state of a query that is presented in the previous chapter. Based on this notion we show that once sufficient state information has been constructed at every partition (i.e., replicating some tuples between processors), queries can be processed in parallel without additional data transfers. We discuss several optimization strategies and present a preliminary quantitative analysis of our approach. Before we proceed, we differentiate two classes of range-partitioning function: *homogeneous* and *heterogeneous*, which are defined below.

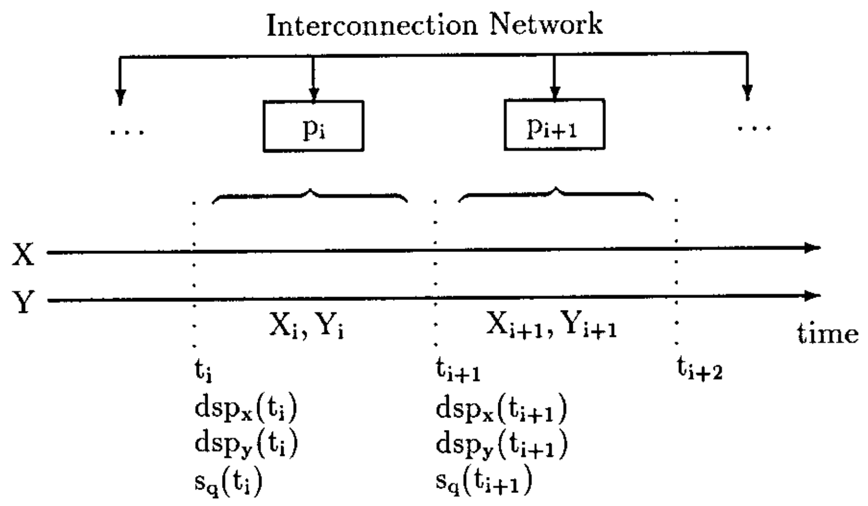


Figure 6.5: Constructing state information at partitioning boundaries

Definition 6.1 Relations are referred to as *homogeneously* range-partitioned if their partitioning functions are identical, i.e., all the partitioning boundaries are identical. If different range-partitioning functions are used, it is referred to as *heterogeneous.* □

In this section, we focus on TSJ_1 join queries whose operand relations are homogeneously range-partitioned.

6.2.1 State Information at Partitioning Boundaries

We now outline the parallel processing strategy for TSJ_1 queries. In Chapter 5, we discuss the notion of periodically checkpointing a temporal query execution along the time dimension. At every checkpoint, we store the checkpoint time, state information and data stream pointers as checkpoints. This notion of periodic checkpointing can be easily applied to parallel query processing in database machines as described below.

The analogy to periodic checkpointing in parallel database machines is that temporal relations are homogeneously range-partitioned on a timestamp (TS or TE), i.e., processor p_i is assigned an interval $[t_i, t_{i+1})$ as depicted in Figure 6.5. One can simply assume that the data stream pointers at the partitioning boundary t_i, $dsp_x(t_i)$ and $dsp_y(t_i)$, are "pointing" at the relation fragments X_i and Y_i respectively. Given a query Q, the strategy is to *construct* sufficient state information at *every* partitioning boundary so that each processor can *independently* process the query Q on its local relation fragments using the constructed state information. For example, as shown in Figure 6.5, processor p_i will process its local fragments X_i and Y_i using the state information $s_q(t_i)$. Similarly, p_{i+1} will process X_{i+1} and Y_{i+1} using the state information $s_q(t_{i+1})$. In general, the strategy has three distinct phases:

Replication Phase Construct sufficient state information for every partition.

Join Phase The query can be executed by each processor using its local relation fragments and the constructed state information.

Merge Phase The query response is produced by merging the results returned from all processors and eliminating duplicates.

Let us emphasize that the TSJ_1 queries are multi-way temporal joins. For the sake of simplicity of exposition, we concentrate on joins of two relations unless otherwise stated.

6.2.2 Replication Phase

In this subsection, we discuss the construction of state information at each partition for a query to be processed in parallel by each processor. Intuitively, the replication phase is to copy tuples whose lifespans intersect with each other such that they co-exist at the same processor for the subsequent join phase. Based on the query qualification, one can derive a *state predicate* for each operand relation[4]. Using the derived state predicates, one may be able to limit which tuples should be copied as we shall see shortly. For a query qualification P(X,Y), a state predicate for relation X can be derived from P(X,Y) by substituting join predicates and comparison predicates that involve the relation Y. Analogously, we can derive a state predicate for relation Y.

By propagating constraints between attributes, one can sometimes find a more restrictive state predicate. For example, consider the query of finding the head

[4] As we discuss in Chapter 5, a state predicate for a relation R, denoted as $P|_r$, is a query qualification on R.

of the studio "MGM" and the directors who worked for the studio at the same time is:

$$\sigma_{\text{intersect-join(Studio,Dir)} \wedge \text{Studio.Sname=Dir.Sname} \wedge \text{Studio.Sname=MGM}}(\text{Studio,Dir}).$$

Using the abovementioned (simplistic) method, the state predicate for the relation Dir is "true" while the state predicate for the relation Studio is "Studio.Sname=MGM". Intuitively, only tuples in relation Dir that satisfy the predicate "Dir.Sname=MGM" would participate in the join and thus only these tuples should be replicated as state information. For completeness, we describe here a mechanism in which bounds on timestamp values can be propagated between relations [Ull82, Chak84, Jar84, She89].

We first consider constraints (i.e., upper and lower bounds) on timestamps of individual relation R: .

$$\text{ts}^- \leq \text{TS} < \text{ts}^+ \text{ and } \text{te}^- \leq \text{TE} < \text{te}^+$$

where ts^-, ts^+, te^- and te^+ are constants. That is, the values of TS timestamp are bounded by the interval $[\text{ts}^-, \text{ts}^+)$ and those of TE are bounded by $[\text{te}^-, \text{te}^+)$. If a particular timestamp is not explicitly constrained, its default constraint interval is $[0, now)$. Since in our data model the TS value must be smaller than the TE value in each tuple, the two constraint intervals are therefore related:

- If $\text{te}^- \leq \text{ts}^-$, the constraint on TE becomes: $\text{ts}^- + 1 \leq \text{TE} < \text{te}^+$.
- If $\text{te}^+ \leq \text{ts}^+$, the constraint on TS becomes: $\text{ts}^- \leq \text{TS} < \text{te}^+ - 1$.

That is, the data values of TS or TE may further be constrained. From now on, without loss of generality, we assume that "$\text{ts}^- < \text{te}^-$" and "$\text{ts}^+ < \text{te}^+$" hold for each individual relation in our discussion. Note that if the lower bound is larger than or equal to the upper bound, the query response must be necessarily null.

The relationship among constraints on timestamps can be easily explained and determined using a constraint graph structure — Algorithm 6.1 can be used to construct a constraint graph G using the query qualification P.

Algorithm 6.1 Constraint Graph Construction[5]:

[5] Note that whether or not P has a comparison predicate that involves the operator "$\neq$"

1. For each timestamp T of an operand relation, there is a node (labeled with T) in the graph G. Each node is tagged with a pair of values representing the upper and lower bound of the constraint on the timestamp data values.

2. For every relation R_i, a solid directed arc from the node R_i.TE to node R_i.TS is added to G.

3. If "$T_1<T_2$" is a predicate in P for any timestamps T_1 and T_2, a solid directed arc from the node T_2 to the node T_1 is added to G. Similarly, a dotted directed arc from T_2 to T_1 is added to G if "$T_1 \leq T_2$" is in P.[6]

4. If "$T_1=T_2$" is a predicate in P for any timestamps T_1 and T_2 of different relations, we merge these nodes together resulting a single node (labeled with $\{T_1,T_2\}$) that represents both timestamps T_1 and T_2. The largest lower bound of nodes T_1 and T_2 becomes the lower bound of the new node $\{T_1,T_2\}$. Similarly, the smallest upper bound of T_1 and T_2 becomes its upper bound.

5. Given the graph G that is constructed so far, we detect if there is a cycle in G. A path from a node T_2 to a node T_1 exists if (1) there is a (solid or dotted) directed arc from T_2 to T_1, or (2) there is a directed arc from T_2 to another node T_3 and there is a path from T_3 to T_1. A cycle exists if there is a path from any node to itself. There are two cases when a cycle exists:

   - The cycle has at least one solid directed arc (which represents "<" relationship). In this case, the user qualification is identically false and thus the query produces a null response.
   - All the directed arcs in the cycle are dotted arcs (which represent "≤" relationship). In this case, all the nodes in the cycle are merged together as in the case of "=" relationship. The lower and upper bounds of the new node are determined accordingly.

Note that for any TSJ_1 query, G is partially ordered. □

and a timestamp (e.g., "R_i.TS $\neq t_1$") does not have any impact on the graph construction algorithm.

[6] In Chapter 4, we note that semantic query optimization can play a significant role in query optimization. Using additional semantic information regarding temporal relationships between timestamps, more arcs may be added in the graph.

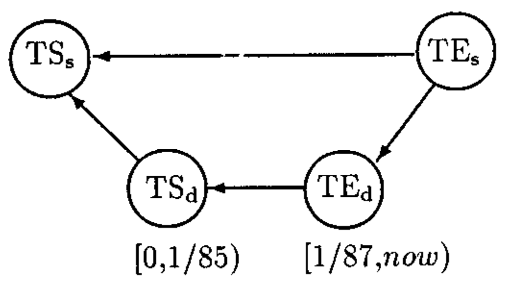


Figure 6.6: Constraint graph for Example 6.1 — upper and lower bounds on timestamps

When the constraint graph is constructed using the Algorithm 6.1, constraints can be propagated between nodes using Algorithm 6.2 below.

Algorithm 6.2 Constraint Propagation: The upper bounds are propagated from roots to leaves whereas the lower bounds are propagated in the opposition direction. Suppose that the constraint on a node T_1 is $[t_1^-, t_1^+)$ and that on a node T_2 is $[t_2^-, t_2^+)$.

1. Dotted arc: for a dotted directed arc from a node T_2 to a node T_1, the propagation of the upper bound of T_2 to T_1 results the new upper bound of T_1 being $\mathsf{min}(t_1^+, t_2^+)$. The propagation of the lower bound of node T_1 to T_2 results the new lower bound of T_2 being $\mathsf{max}(t_1^-, t_2^-)$.
2. Solid arc: If the arc from T_2 to T_1 is solid, the new upper bound of node T_1 is $\mathsf{min}(t_1^+, t_2^+ - 1)$. The propagation of the lower bound of T_1 to T_2 results the new lower bound of T_2 being $\mathsf{max}(t_1^-, t_2^- + 1)$.

□

Example 6.1 Consider the film industry examples. Suppose we want the combinations of all stars and directors such that the star acted in films directed by the director during the entire period of time in which the director worked for a studio for the entire interval [1/85,12/86). The query is:

$$\sigma_{\text{contain-join(Stars,Dir)} \wedge \text{Dir.TS}<1/85 \wedge 12/86<\text{Dir.TE}}(\text{Stars,Dir}).$$

The constraints on timestamp values are represented by a constraint graph as shown in the Figure 6.6. A node represents a timestamp and a solid arrow represents the "before" (i.e., <) relationship between two timestamps. The values of timestamps "Dir.TS" and "Dir.TE" are bounded by [0,1/85) and [1/87,*now*) respectively[7].

The constraints on timestamp values are then propagated among nodes. For example, in Figure 6.6, the TS values of relation Stars (i.e., TS_s) are bounded by the interval [0,12/84) while the TE values (i.e., TE_s) are bounded by [2/87,*now*). Thus, the state predicate for the relation Stars becomes "Stars.TS<12/84 $\wedge$ 2/87$\leq$Stars.TE". □

Given a query $Q \equiv \sigma_P(X,Y) \in TSJ_1$, we first derive a state predicate for each operand with the results being denoted by $P|_x$ and $P|_y$. Using these state predicates, one can construct the state information on each processor which is defined as follows.

Definition 6.2 Given that a partitioning interval $[t_i,t_{i+1})$ is assigned to processor p_i, the *state information* for a relation R at the partitioning boundary t_i, denoted as $s_r(t_i)$, contains:

$\{ r \mid r \in R \wedge r.TS<t_i \wedge t_i \leq r.TE \wedge P|_r(r) \}$ if R is TS range-partitioned

$\{ r \mid r \in R \wedge r.TS<t_{i+1} \wedge t_{i+1} \leq r.TE \wedge P|_r(r) \}$ if R is TE range-partitioned

where $P|_r$ is the derived state predicate for the relation R, and $P|_r(r)$ holds for the tuple r. □

Essentially, all qualified tuples (based on the state predicate) whose lifespan intersects with the partitioning interval $[t_i, t_{i+1})$ and are not stored in the local fragments at processor p_i will be replicated at processor p_i as the "state information". As soon as the state information for all operand relations at all partitions have been constructed, the join phase, which is the focus of the following subsection, can proceed.

[7] We assume that the time granularity in this example is "month", i.e., consecutive months are mapped into consecutive integers.

6.2.3 Join Phase and Merge Phase

For a query $Q \equiv \sigma_P(X,Y) \in TSJ_1$, each processor p_i can execute Q using its local relation fragments and the state information constructed at p_i. The response to Q is the union of the results (eliminating duplicates) from all processors:

$$\bigcup_{1 \leq i \leq n_{pi}} \{ \sigma_P(X_i, s_y(t_i)) \cup \sigma_P(Y_i, s_x(t_i)) \cup \sigma_P(X_i, Y_i) \}$$

where n_{pi} is the total number of partitions. For a TSJ_1 join query involving m relations, we can use the following strategy:

$$\bigcup_{1 \leq i \leq n_{pi}} \{ \sigma_P((R_{1,i} \cup s_{r_1}(t_i)), \cdots, (R_{m,i} \cup s_{r_m}(t_i))) \}$$

where $R_{j,i}$ is the ith fragment (i.e., partition $[t_i, t_{i+1})$) of the relation R_j, $1 \leq j \leq m$, which is stored at processor p_i. In other words, the local join for each partition is this: for each relation R_j, we "merge" its state information (i.e., $s_{r_j}(t_i)$) with its local fragment (i.e., $R_{j,i}$), and then join all the newly "merged" fragments.

We sketch the proof of the correctness of the parallel join strategy for TSJ_1 queries as follows:

> Given a query $Q \equiv \sigma_P(R_1, \cdots, R_m) \in TSJ_1$. By the definition of TSJ_1, each m-tuple $<r_1, r_2, \cdots, r_m>$, where $r_k \in R_k$ for $1 \leq k \leq m$, that satisfies the join condition must have a common time point, denoted as t_c, as illustrated in Figure 6.7. That is, for each participating tuple $r_k \in R_k$, r_k satisfies the derived state predicate $P|_{R_k}$ (otherwise r_k will not be a component of the m-tuple $<r_1, r_2, \cdots, r_m>$). Without loss of generality, we assume that t_c falls into a partition $[t_i, t_{i+1})$ which is assigned to a processor p_i [8]. Specifically, "$t_i \leq t_c \wedge t_c < t_{i+1}$" holds. For every participating tuple, $r_k \in R_k$, $1 \leq k \leq m$, exactly one of the following conditions must hold:
>
> 1. The relation R_k is TS range-partitioned and the tuple r_k starts during the partition $[t_i, t_{i+1})$. That is, r_k is a tuple in the relation fragment stored at processor p_i.

[8] It is possible that the operand tuples share a common time interval that spans multiple partitions. In this case, the m-tuple may be produced by more than one processor. The final merge phase would eliminate the duplicates.

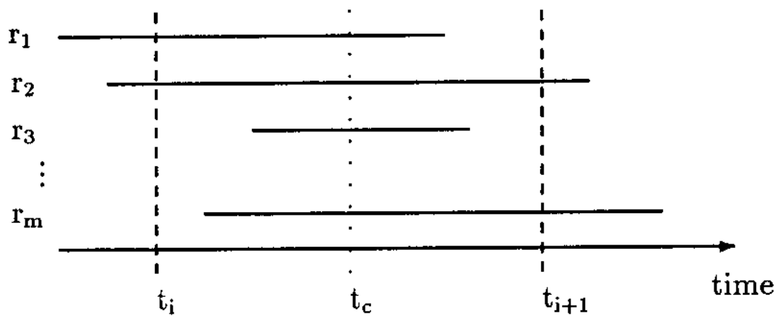


Figure 6.7: A m-tuple $<r_1, r_2, \cdots, r_m>$ that satisfies a TSJ_1 join condition

2. The relation R_k is TS range-partitioned and the tuple r_k starts earlier than t_i and span the partitioning boundary t_i. That is, r_k should have been replicated at processor p_i as the state information.
3. The relation R_k is TE range-partitioned and the tuple r_k ends during the partition $[t_i, t_{i+1})$. That is, r_k is a tuple in the relation fragment stored at processor p_i.
4. The relation R_k is TE range-partitioned and the tuple r_k ends at or later than t_{i+1} and span the partitioning boundary t_{i+1}. That is, r_k should have been replicated at processor p_i as the state information.

Therefore, each component in the m-tuple $<r_1, r_2, \cdots, r_m>$ can be found either in the state information or in the local fragment. Hence, the m-tuple $<r_1, r_2, \cdots, r_m>$ is produced by the processor p_i. In other words, every m-tuple in the query response is produced by at least a processor in the parallel join processing strategy. Q.E.D.

6.2.4 Optimization: Reducing State Information

The definition of the state information of a relation at a partition as qualified tuples (based on the derived state predicates) that span the partition is general enough to support the parallel query processing strategy for all queries in TSJ_1

whose operand relations are homogeneously range-partitioned. In this subsection, we discuss some optimization opportunities in which the number of tuples replicated as state information can be reduced.

6.2.4.1 Asymmetry Property

First, we define the *asymmetry* property of operands in a TSJ_1 join query with respect to the TS and TE timestamps.

Definition 6.3 Given a query $Q \equiv \sigma_{P(R_1,\cdots,R_m)}(R_1,\cdots,R_m) \in TSJ_1$. The relation R_k, $k \in \{1,\cdots,m\}$, has the asymmetry property with respect to the *TS* timestamp if the following condition is satisfied:

$$P(R_1,\cdots,R_m) \Rightarrow R_k.TS \geq R_i.TS, \qquad \forall \ 1 \leq i \leq m. \qquad \square$$

Definition 6.4 Given a query $Q \equiv \sigma_{P(R_1,\cdots,R_m)}(R_1,\cdots,R_m) \in TSJ_1$. The relation R_k, $k \in \{1,\cdots,m\}$, has the asymmetry property with respect to the *TE* timestamp if the following condition is satisfied:

$$P(R_1,\cdots,R_m) \Rightarrow R_k.TE \leq R_i.TE, \qquad \forall \ 1 \leq i \leq m. \qquad \square$$

For each m-tuple $<r_1,\cdots,r_m>$ that satisfies the query qualification P, where $r_i \in R_i$ for $1 \leq i \leq m$, the asymmetry property with respect to the TS timestamp means that the tuple r_k must have the maximal TS value among all participating tuples. For example, consider contain-join(X,Y) whose join condition is "X.TS<Y.TS $\wedge$ Y.TE<X.TE" and overlap-join(X,Y) whose join condition is "X.TS<Y.TS $\wedge$ Y.TS<X.TE $\wedge$ X.TE<Y.TE". The relation Y in both contain-join(X,Y) and overlap-join(X,Y) has the asymmetry property with respect to the TS timestamp. Similarly, the asymmetry property with respect to the TE timestamp means that the tuple r_k must have the minimal TE value among all participating tuples. For example, the relation Y in contain-join(X,Y) and the relation X in overlap-join(X,Y) have this asymmetry property.

Depending on whether a relation is TS or TE range-partitioned, the asymmetry properties can be used to show that constructing the state information for some relation is redundant, and therefore the replication phase for that particular relation can be eliminated.

Theorem 6.1 Redundant State Information.

Given:

- a query $Q \equiv \sigma_P(R_1, \cdots, R_m) \in TSJ_1$, and
- there are m', where $1 \leq m' \leq m$, relations which have the asymmetry property with respect to their partitioning timestamp (we use a subscript j to denote these relations as R_j where $j = \{1, \cdots, m'\}$).

Conditions under which the state information for a relation is redundant are:

- All R_j's, $j = \{1, \cdots, m'\}$, are TS range-partitioned. Then the state information of all R_j's are redundant.
- All R_j's, $j = \{1, \cdots, m'\}$, are TE range-partitioned. Then the state information of all R_j's are redundant.
- Some R_j's are TS range-partitioned while others are TE range-partitioned. That is, R_j's can be partitioned into two disjoint sets:

  $R_j|_{TS}$ and $R_j|_{TE}$.

  The first set corresponds to TS range-partitioning while the second set corresponds to TE range-partitioning. Then relations in either set have the redundant state information property[9]. □

Proof We now sketch the proof for the Theorem 6.1 as follows:

We consider the case of the TS range-partitioning; the argument for other cases is similar. Given a TSJ_1 join query, all the components of each m-tuple $<r_1, \cdots, r_m>$, where $r_i \in R_i$ for $1 \leq i \leq m$, that satisfies the join condition must have a common time point. Also, given that

[9] Then we have a choice of selecting which relations to have the redundant state information property.

relation R_k, $k \in \{1, \cdots, m\}$, has the asymmetry property, the component r_k in the m-tuple must have the largest TS value and therefore the r_k.TS value must also be a common time point. Since the relation R_k is TS range-partitioned, the m-tuple must have been produced in the partition where the tuple r_k is stored. Hence, the tuple r_k need not be replicated to other partitions for the join process. Q.E.D.

There are several interesting observations that can be made. First, when all temporal join predicates are inequalities, only *one* operand relation has the redundant state information property. Second, for contain-join(X,Y) the relation Y has the asymmetry property with respect to both the TS and TE timestamps. For this reason, state information for the relation Y need not be constructed regardless of whether the relation is TS or TE range-partitioned. Thirdly, when there is an *equality* temporal join predicate (e.g., "X.TS=Y.TS" or "X.TE=Y.TE") between two relations, and both relations have the asymmetry property with respect to their join attribute (i.e., timestamp), Y has the redundant state information property if the state information of X is redundant (or vice versa). As another example, consider meet-join(X,Y), whose join condition is "X.TE=Y.TS", and X is TE range-partitioned while Y is TS range-partitioned. Both relations have the asymmetry property with respect to the partitioning timestamp, and thus the state information for both X and Y are redundant.

Example 6.2 Consider the overlap-join(X,Y) whose join condition is "X.TS< Y.TS $\wedge$ Y.TS< X.TE $\wedge$ X.TE<Y.TE". Suppose that the relation Y is TS range-partitioned. Only the state information of the relation X (but not relation Y) has to participate in the join phase. Therefore one has to replicate only tuples of relation X as state information, and the construction phase of state information of the relation Y can be eliminated. □

Example 6.3 Consider Example 6.1. The join condition is "Stars.TS<Dir.TS $\wedge$Dir.TE<Stars.TE", i.e., "contain-join(Stars, Dir)". If the relation Dir is TS range-partitioned, its construction phase of state information can be eliminated. That is, one has to replicate only tuples of relation Stars as state information. □

6.2.4.2 Use of Statistics

We show how several simple statistics on the data in each partition can further reduce data movement and discuss their pros and cons. Again, for the sake of simplifying our discussion, we focus on joins of two relations — X and Y, assuming both relations are TS range-partitioned.

Suppose that the database system keeps the maximum and minimum of the TS and TE values for every relation fragment. For example, the TS and TE values of a relation fragment Y_i (i.e., the partition $[t_i,t_{i+1})$) of the relation Y are bounded by the intervals: $[Y_i.TS_{min}, Y_i.TS_{max})$ and $[Y_i.TE_{min}, Y_i.TE_{max})$ respectively. We further suppose that the fragment Y_i is stored at processor p_i. Together with the query qualification, the statistics can be used to further reduce data replication of the relation X. To illustrate this point, we consider the following examples:

- Consider the overlap-join(X,Y) whose join condition is "$X.TS<Y.TS \wedge Y.TS<X.TE \wedge X.TE <Y.TE$". Intuitively, tuples in the relation X that span the partitioning boundary t_i and whose TE values are smaller than or equal to $Y_i.TS_{min}$ need not be sent to processor p_i because these X tuples do not join with any tuples in Y_i. This is also true for X tuples that span t_i and whose TE values are larger than or equal to $Y_i.TE_{max}-1$.

- Consider the contain-join(X,Y) whose join condition is "$X.TS<Y.TS \wedge Y.TE<X.TE$". Tuples in the relation X that span t_i and whose TE values are smaller than or equal to $Y_i.TE_{min}$ need not be sent to p_i as state information.

- Consider the meet-join(X,Y) whose join condition is "$X.TE=Y.TS$". Tuples in the relation X that span t_i and whose TE values are smaller than $Y_i.TS_{min}$ or larger than $Y_i.TS_{max}$ need not be sent to p_i as state information.

To eliminate redundant tuples from being replicated as state information, one can make use of the constraint propagation algorithm that is presented earlier. For example, consider the overlap-join(X,Y) whose constraint graph is shown in Figure 6.8. After the upper and lower bounds have been propagated, the TS values of relation X are bounded by the interval $[0,Y_i.TS_{max}-1)$ while the

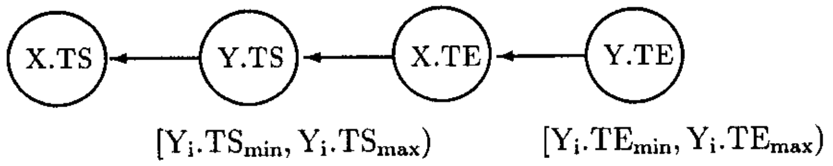

X.TS
Y.TS
X.TE
Y.TE
[Y_i.TS$_{min}$, Y_i.TS$_{max}$)
[Y_i.TE$_{min}$, Y_i.TE$_{max}$)

Figure 6.8: Overlap-join(X,Y): "X.TS<Y.TS $\wedge$ Y.TS<X.TE $\wedge$ X.TE<Y.TE"

TE values are bounded by [$Y_i.TS_{min}+1, Y_i.TE_{max}-1$). The state information of relation X at the partitioning boundary t_i therefore contains:

$$\{\ x \mid x \in X \wedge x.TS<t_i \wedge t_i \leq x.TE \wedge P|_x(x) \\ \wedge\ x.TS<Y_i.TS_{max}-1 \wedge Y_i.TS_{min}<x.TE \wedge x.TE<Y_i.TE_{max}-1\ \}$$

where the derived state predicate $P|_x$ is actually "true" for the overlap-join(X,Y). Note that the predicate "$x.TS<Y_i.TS_{max}-1$" is subsumed because "$x.TS<t_i$" and "$t_i \leq Y_i.TS_{max}$" hold.

The tradeoffs for these optimizations include that keeping these statistics consistent for every relation fragment incurs some overhead. For example, suppose the relation Y in the above example (see Figure 6.8) is TE range-partitioned, i.e., the state information of relation Y has to be constructed. This means that the four statistics ($Y_i.TS_{max}$, $Y_i.TE_{max}$, $Y_i.TS_{min}$ and $Y_i.TE_{min}$) do not always reflect the actual bounds on the timestamp values. Therefore the actual bounds have to be computed after the state information of relation Y has been completely constructed, and the results have to be broadcast to all senders (i.e., the processors which send tuples X). This may require substantial coordination overhead between processors. Moreover, determine if a tuple should be sent to a processor requires the evaluation of a more complex qualification. However, for the operand relation that has the property of redundant state information discussed earlier, the four statistics of this relation do represent the actual bounds.

6.2.5 Participant Processors

For the parallel processing strategies that are discussed earlier, *all* processors participate in the replication and join phases. However, for some TSJ_1 queries, it can be determined a priori that some processors necessarily return a null response when they perform the local join. Similarly, it can also be determined a priori that some processors need not replicate some fragments of a relation in the replication phase (in the sense that the relation fragments will not contribute to the query response). These situations may occur when the user query qualification contains some comparison predicates involving timestamps (such as in snapshot or interval queries). To illustrate the idea, we first define the notion of *replication-interval* and *join-interval.*

Definition 6.5 The *replication-interval* for an operand relation in a query is defined as the minimal interval with the property that only tuples whose partitioning timestamp (i.e., TS or TE) value falls within the interval can possibly participate as state information[10]. □

Definition 6.6 The *join-interval* for a query is defined as the minimal interval with the property that only tuples whose partitioning timestamp value falls within the interval can possibly contribute to the query response. □

Definition 6.7 A *join processor* is referred to as a processor that has to participate in the join phase (of our parallel processing strategy), i.e., the processor which has a partitioning interval that intersects with the join-interval. Otherwise, it is referred to as a non-join processor which necessarily returns a null response. □

Definition 6.8 A *replication processor* is referred to as a processor that has to participate in the replication phase (of our parallel processing strategy), i.e., the processor which has a partitioning interval that intersects with the replication-intervals. Otherwise, it is referred to as a non-replication processor. □

[10] Note that if the relation has the property of redundant state information discussed in the previous section, the corresponding replication-interval is necessarily null (i.e., no tuples will be replicated as state information).

If the join-interval is null, the join response is necessarily null. Similarly, if the replication-interval for a relation is null, tuples of that relation need not be replicated as state information. Otherwise, tuples in the replication-intervals are replicated on join processors as state information for the join phase.

The join-interval and replication-intervals for a given query depend on the following:

1. the TS and TE range-partitioning functions, and
2. the query qualification:
    - the relationship between the comparison predicates involving timestamps and the temporal join predicates.
    - the property of redundant state information discussed earlier.

Earlier we address the issue of determining the upper and lower bounds on the TS and TE values of each individual relation by propagating constraints between relations using a constraint graph. Below we address other issues; we first define the I-intersect operator that will be used in the remainder of this section.

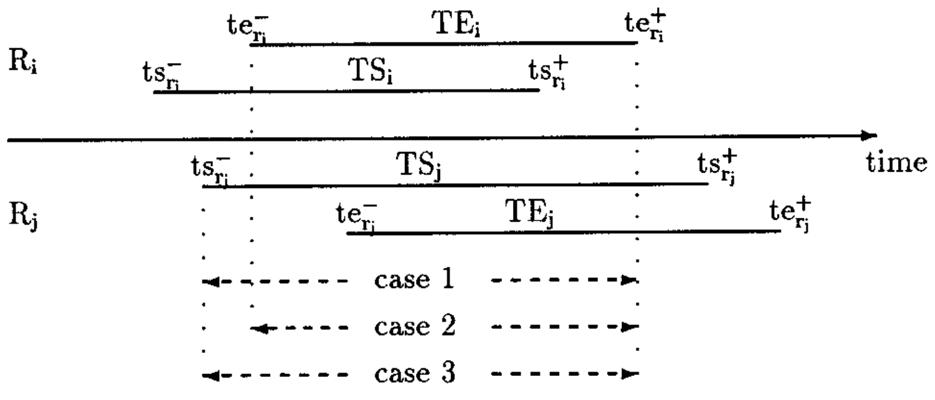
Definition 6.9 Given two non-null intervals $[ts_1, te_1)$ and $[ts_2, te_2)$, we define:

$$\text{I-intersect}([ts_1, te_1),[ts_2, te_2)) \equiv [\max(ts_1, ts_2),\ \min(te_1, te_2))^{11}.$$

That is, the time interval that the I-intersect operator produces is the intersection of the two operand intervals. If the left-end value of an interval is larger than or equal to the right-end value, the resultant interval is equivalent to a null interval. □

The impact of TS and TE range-partitioning schemes on join-intervals and replication-intervals is as follows. Consider a TSJ_1 join query with relations R_i and R_j. Suppose that the upper and lower bounds of the timestamp values have been determined — where ts_r^- and ts_r^+ represent the lower and upper bounds on the TS timestamp of relation R respectively, and similarly te_r^- and te_r^+ for the TE timestamp as illustrated in Figure 6.9. There are three cases to be considered depending how R_i and R_j are (TS or TE) range-partitioned.

[11] The operator max(A,B) (respectively min(A,B)) returns the larger (respectively smaller) value of A and B.



Constraints:
$ts^-_{r_i} \leq TS_i < ts^+_{r_i}$ $\quad$ $ts^-_{r_j} \leq TS_j < ts^+_{r_j}$
$te^-_{r_i} \leq TE_i < te^+_{r_i}$ $\quad$ $te^-_{r_j} \leq TE_j < te^+_{r_j}$
$TS_i < TE_i$ and $TS_j < TE_j$ for all tuples

Figure 6.9: Determining the join-interval and the replication-intervals

Case 1: R_i and R_j are TS range-partitioned

There are three situations to be considered:

1. When neither R_i nor R_j has the property of redundant state information, the join-interval for the query is:

$$[\max(ts^-_{r_i}, ts^-_{r_j}),\ \min(\max(ts^+_{r_i}, ts^+_{r_j}),\ te^+_{r_i}, te^+_{r_j})).$$

The replication-interval for relation R_k, where $k \in \{i,j\}$, is:

$$\text{I-intersect}(\ [\min(ts^-_{r_i}, ts^-_{r_j}),\ \min(ts^+_{r_i}, ts^+_{r_j})),\ [ts^-_{r_k}, ts^+_{r_k})\ ).$$

In Figure 6.9, the join-interval is $[ts^-_{r_j}, te^+_{r_i})$, and the replication-interval for R_i is $[ts^-_{r_i}, ts^+_{r_i})$ while that of R_j is $[ts^-_{r_j}, ts^+_{r_i})$.

2. When the relation R_i (but not R_j) has the property of redundant state information, the join-interval becomes:

$$[\max(ts^-_{r_i}, ts^-_{r_j}),\ \min(ts^+_{r_i}, te^+_{r_j})),$$

and the replication-interval for R_i is a null interval while that of R_j is:

$[ts^-_{r_j}, \min(ts^+_{r_i}, ts^+_{r_j}))$.

3. When both relations R_i and R_j have the property of redundant state information, the join-interval becomes:

   $\textsf{I-intersect}(\ [ts^-_{r_i}, ts^+_{r_i}),\ [ts^-_{r_j}, ts^+_{r_j})\ )$[12]

   and the replication-intervals of both relations are null intervals.

Example 6.4 Find the directors who joined a studio sometime during the interval [1/85,1/86) and also became the head of the studio sometime during [1/86,1/87):

$$\sigma_{1/85 \leq Dir.TS \wedge Dir.TS < 1/86 \wedge 1/86 \leq Studio.TS \wedge Studio.TS < 1/87 \wedge P}\ (Studio, Dir)$$

where P is "intersect-join(Studio,Dir) $\wedge$ Studio.Sname = Dir.Sname". Suppose that both relations Studio and Dir are TS range-partitioned. The constraints on the TS timestamps of both relations are illustrated in Figure 6.10. The join-interval for the query is [1/86,1/87). The replication-interval for relation Dir is [1/85,1/86) and that for relation Studio is a null interval. That is, the state predicates for relations Dir and Studio are "$1/85 \leq$ Dir.TS $\wedge$ Dir.TS $< 1/86 \wedge$ Dir.TE $\geq 1/86$" and "false" respectively. The join phase involves only joining the state information for the relation Dir and the local fragments of relation Studio that start during [1/86,1/87)[13]. □

Case 2: <u>R_i and R_j are TE range-partitioned</u>

When both relations are TE range-partitioned, the join-interval and replication-intervals are simply "mirror-images" of those for the above TS range-partitioning case. To simplify our presentation, we consider only the situation when both relations do not have the property of redundant state information. That is, the join-interval is:

[12] Note that there must be an equality join predicate "R_i.TS=R_j.TS" in the query qualification.

[13] One can obtain tighter bounds by examining (and thus accessing) local fragment of relation Studio that start during [1/86,1/87), as well as further analyzing the temporal join operators involved.

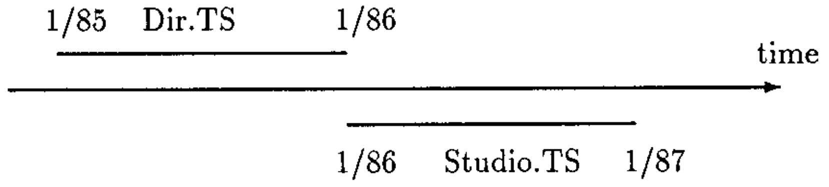


Figure 6.10: Constraints on TS timestamps of relations Dir and Studio

$$[\max(\min(te^-_{r_i}, te^-_{r_j}), ts^-_{r_i}, ts^-_{r_j}), \min(te^+_{r_i}, te^+_{r_j})).$$

In Figure 6.9, the join-interval is $[te^-_{r_i}, te^+_{r_i})$. The replication-interval of R_k, where k is either i or j, is:

$$\text{I-intersect}(\ [\max(te^-_{r_i}, te^-_{r_j}), \max(te^+_{r_i}, te^+_{r_j})), [te^-_{r_k}, te^+_{r_k})\ ).$$

Case 3: R_i is TS range-partitioned and R_j is TE range-partitioned

1. When neither R_i nor R_j has the property of redundant state information, the join-interval for the query is:

   $$[\max(ts^-_{r_i}, ts^-_{r_j}), \min(te^+_{r_i}, te^+_{r_j})).$$

   In Figure 6.9, the join-interval is $[ts^-_{r_i}, te^+_{r_j})$. The replication-interval of R_i is:

   $$[ts^-_{r_i}, \min(te^+_{r_j}, ts^+_{r_i}))$$

   while that of R_j is:

   $$[\max(ts^-_{r_i}, te^-_{r_j}), te^+_{r_j}).$$

2. When the relation R_i (but not R_j) has the property of redundant state information, the join-interval becomes:

   $$[\max(ts^-_{r_i}, ts^-_{r_j}), \min(ts^+_{r_i}, te^+_{r_j})).$$

   The replication-interval of R_i is a null interval while that of R_j is:

   $$[\max(ts^-_{r_i}, te^-_{r_j}), te^+_{r_j}).$$

3. When the relation R_j (but not R_i) has the property of redundant state information, the join-interval becomes:

$[\max(ts^-_{r_i}, te^-_{r_j}), \min(te^+_{r_i}, te^+_{r_j}))$.

The replication-interval of R_j is a null interval while that of R_i is:

$[ts^-_{r_i}, \min(te^+_{r_j}, ts^+_{r_i}))$.

4. When both relations have the property of redundant state information, the join-interval becomes:

   $\text{I-intersect}(\ [ts^-_{r_i}, ts^+_{r_i}),\ [te^-_{r_j}, te^+_{r_j})\ )$.

   The replication-intervals are null intervals.

In summary, the above syntactic mechanism can be used to determine which processors have to send data as state information and which processors have to receive data as state information. One may wonder if we can move tuples from join-interval to replication-interval for the join phase. In general, all tuples in the join-interval would have to be replicated to all partitions in the replication-intervals which is very expensive and should be avoided. In the following subsection, we discuss the overhead of constructing the state information — the analysis is based on the approach presented in Chapter 5.

6.2.6 Quantitative Analysis

We present a first-cut quantitative analysis on the overhead associated with constructing state information of a relation. The overhead associated with constructing the state information for a relation R can be measured in terms of the number of tuples to be replicated since the communication and/or storage costs will be directly related to this number (and the tuple size). We let λ be the rate of insertion of tuples into the relation R, $\overline{T_{ls}}$ be the average tuple lifespan, and TR_{ls} be the relation lifespan. Using Little's result [Lit61], the average number of tuples that are active as of a particular time, denoted by $\overline{n}$, is given by:

$$\overline{n} \ = \ \lambda \cdot \overline{T_{ls}}.$$

A natural assumption is that the average number of active tuples at partitioning boundaries is also $\overline{n}$. Similarly, the total number of tuples in the relation R is:

$$\lambda \cdot TR_{ls}.$$

Suppose that the selectivity of the state predicate q that is used to construct the state information for the relation R is σ_q and is defined as the fraction of tuples in R that satisfy q. The number of tuples that are copied as state information is then given by:

$$\sigma_q \cdot n_p \cdot \overline{n} \;=\; \sigma_q \cdot n_p \cdot \lambda \cdot \overline{T_{ls}}$$

where n_p is the number of partitions at which state information has to be constructed (i.e., the partitioning intervals that overlap with the join-interval discussed in the previous section). Note that n_p must be smaller than the total number of partitioning intervals (n_{pi}).

Definition 6.10 The *overhead* is defined as the ratio of the number of tuples to be copied over the total number of tuples in the relation:

$$\sigma_q \cdot n_p \cdot \lambda \cdot \overline{T_{ls}}/(\lambda \cdot TR_{ls}) \;=\; \sigma_q \cdot n_p \cdot \overline{T_{ls}}/TR_{ls}. \qquad \square$$

The quantity is consistent with our intuition that:

- n_p: the overhead increases as the number of partitions with state information increases.
- σ_q: the more selective the state predicate (which constructs the state information) is, the less overhead is incurred. For this reason, one should derive more restrictive state predicates for operand relations.
- $\overline{T_{ls}}/TR_{ls}$: the overhead is smaller for relations with relatively short tuple lifespans (compared with the relation lifespan).

Note that the above analysis does not depend on whether the time-varying attribute is continuous or non-continuous. In case of continuous time-varying attribute, the average number of active tuples at a particular time is:

$$\overline{n} \;=\; \lambda \cdot \overline{T_{ls}} \;=\; \overline{T_{ls}} \cdot (\overline{Ns}/\overline{T_{ls}}) \;=\; \overline{Ns}$$

where $\overline{Ns}$ is the average number of surrogates in the relation.

6.3 Implementation Issues

In this section, we discuss several implementation issues that are pertinent to the TS and TE range-partitioning schemes, including issues involving storing state information statically and late updates. We also discuss the application of these range-partitioning schemes to situations where the access patterns to current and history data are different, and when temporal data are modeled as non first normal form relations. Finally, we discuss the extension of the notion of state information for aggregate functions.

6.3.1 Continuously Expanding Time Dimension

Although we regard time points as natural numbers, a major difference between the time domain and an integer domain (such as department number) is that the time dimension is continuously expanding. Moreover, it is advancing in one direction, i.e., the current time is getting larger. This leads to some design issues as described below.

For the range-partitioning function presented earlier, the processor $p_{n_{pi}}$ stores tuples of the last partition $[t_{n_{pi}},now)$. Assume that, for the moment, tuples are evenly distributed among all processors. As time evolves, more tuples with start time (or end time) greater than $t_{n_{pi}}$ are inserted at processor $p_{n_{pi}}$. Eventually, the workload at processor $p_{n_{pi}}$ will be much higher than other processors. The problem is that time is continuously advancing and we have a *fixed* number of partitions. We now briefly discuss several solutions:

Variable number of partitions This approach basically splits the last partition into one or more intervals. That is, the total number of partitions increases as time advances.

Dynamic splitting This is based on balancing the number of tuples kept at each processor. Suppose that each processor should keep about the same number of tuples. When the number of tuples in $p_{n_{pi}}$ exceeds this threshold, the last partition $[t_{n_{pi}},now)$ is then split into two disjoint intervals — $[t_{n_{pi}},t_{n_{pi}+1})$ and $[t_{n_{pi}+1},now)$. The latter interval is assigned to any processor, denoted as $p_{n_{pi}+1}$; a simple approach, called "round-robin" [DeW90], is to choose processor p_j where j equals n_{pi}+1 modulo

n. Tuples which belong to the partition $[t_{n_{pi}+1}, now)$ are moved from $p_{n_{pi}}$ to $p_{n_{pi}+1}$.

Static splitting This is based on static range-partitioning. The idea is to choose the partitioning intervals a priori such that the last partitioning boundary $t_{n_{pi}}$ is large enough that no update time would exceed this value. As time evolves to a certain point, we split the last partition $[t_{n_{pi}}, now)$ into one or more intervals — $[t_{n_{pi}}, t_{n_{pi}+1})$, $[t_{n_{pi}+1}, t_{n_{pi}+2})$, $\cdots$, $[t_{n_{pi}+j}, now)$. These new intervals are then assigned to processors accordingly. This kind of static splitting has the drawback that tuples may not be evenly distributed among processors, but re-organization (as in dynamic splitting) seldom takes place, if it ever does.

Recall that the overhead of constructing state information of a relation at all partitioning boundaries is:

$$\mathrm{Ov} = \sigma_q \cdot (n_{pi} - 1) \cdot \overline{T_{ls}} / TR_{ls}$$

where σ_q is the selectivity of the state predicate q, n_{pi} is the total number of partitions, $\overline{T_{ls}}$ is the average tuple lifespan, and TR_{ls} is the relation lifespan. The rate of change of the overhead is given by:

$$\frac{d\,\mathrm{Ov}}{dt} = \sigma_q \cdot \overline{T_{ls}} \cdot \frac{TR_{ls} \cdot \frac{d\,n_{pi}}{dt} - (n_{pi}-1) \cdot \frac{d\,TR_{ls}}{dt}}{TR_{ls}^2}.$$

The rate of change remains constant if the numerator equals 0. That is,

$$TR_{ls} \cdot \frac{d\,n_{pi}}{dt} - (n_{pi} - 1) \cdot \frac{d\,TR_{ls}}{dt} = 0.$$

Note that the relation lifespan is continuously advancing and the rate of change of the relation lifespan (i.e., $\frac{d\,TR_{ls}}{dt}$) is 1. To obtain an intuitive interpretation, we let the current relation lifespan be:

$$TR_{ls} = (n_{pi} - 1) \cdot T_{ls}.$$

If we create a new partitioning boundary (i.e., $\frac{d\,n_{pi}}{dt}$) at a rate of $1/\overline{T_{ls}}$ (i.e., the average length of partitioning intervals is $\overline{T_{ls}}$), the overhead (Ov) will remain the same. On the other hand, the overhead increases if we create partitioning boundaries faster. Again, this is consistent with our intuition that for shorter partitioning intervals, more qualified tuples may span multiple partitions and therefore the overhead increases. As a rule of thumb, the length of partitioning interval should be greater than the average tuple lifespan. Note that if we denote the average length of partitioning intervals as $\overline{T_{pt}}$, the average number of tuples in a partition is given by:

$\lambda \cdot \overline{T_{pt}}$.

Fixed number of partitions This approach is to dynamically re-adjust the partitioning function to maintain a fixed number of partitions. The idea is to split the last partition $[t_{n_{pi}}, now)$ into $[t_{n_{pi}}, t_{n_{pi}+1})$ and $[t_{n_{pi}+1}, now)$, choose a particular processor p and relocate its tuples to its neighbor(s), and the partition $[t_{n_{pi}+1}, now)$ is assigned to processor p. We discuss two alternatives to relocate tuples:

2-processors Choose two processors, p_i and p_{i+1}, whose partitioning intervals are $[t_i, t_{i+1})$ and $[t_{i+1}, t_{i+2})$ respectively, and move tuples that belong to the partition $[t_i, t_{i+1})$ from p_i to p_{i+1}. Processor p_i is responsible for the partition $[t_{n_{pi}+1}, now)$ while p_{i+1} is responsible for the merged partition $[t_i, t_{i+2})$. Note that the number of tuples in p_{i+1} is roughly twice as many as it was before the re-adjustment.

3-processors Choose processors p_i, p_{i+1} and p_{i+2} whose partitioning intervals are $[t_i, t_{i+1})$, $[t_{i+1}, t_{i+2})$ and $[t_{i+2}, t_{i+3})$ respectively, and move half of the tuples of the partition $[t_{i+1}, t_{i+2})$ to p_i and the rest to p_{i+2}. That is, the partitions for p_i and p_{i+2} are $[t_i, t_{i'})$ and $[t_{i'}, t_{i+3})$ respectively, where $t_{i'}$ is a time point between t_{i+1} and t_{i+2}. Processor p_{i+1} is responsible for the new partition $[t_{i+1}, now)$. Note that the numbers of tuples in p_i and p_{i+2} increase roughly by half.

There are several tradeoff factors. First, dynamic re-adjustment requires moving tuples between processors and therefore the frequency of re-adjustment and the associated overhead should be kept minimal. For example, in the 2-processors scheme, re-adjustment probably occurs less frequently as the fragment size doubles after the re-adjustment. Second, tuples should be more evenly distributed after the re-adjustment. For example, the 3-processors scheme may produce a more even distribution of tuples as the fragment is split into two halfs. In spite of potentially expensive re-adjustment overhead, there are some advantages. Firstly, the number of partitions is fixed. Consequently, less state information will have to be constructed during the processing of complex temporal queries. Moreover, in the other two schemes, the number of partitions becomes larger as time evolves, and eventually some kind of re-organization is needed to keep these partitions manageable (and thus accessing the range-partitioning function is efficient) [DeW90]. The dynamic re-adjustment scheme avoids this situation.

6.3.2 Storing State Information Statically

As opposed to dynamically constructing state information, an alternative is to statically store the state information. One approach is to pick a frequent query whose query qualification is P_x for a relation X[14], and use P_x to create and store the state information at each partitioning boundary. That is, the static state information for relation X at the partitioning boundary t_i contains[15]:

$$\{\ x \mid x \in X \wedge x.TS < t_i \wedge t_i \leq x.TE \wedge P_x(x)\ \}\ \text{if X is TS range-partitioned}$$

$$\{\ x \mid x \in X \wedge x.TS < t_{i+1} \wedge t_{i+1} \leq x.TE \wedge P_x(x)\ \}\ \text{if X is TE range-partitioned}$$

Similarly, we create and statically store the state information of the relation Y using a predicate P_y. In order to process a user query $Q \equiv \sigma_P(X,Y) \in TSJ_1$, we require that "$P|_x \Rightarrow P_x$" and "$P|_y \Rightarrow P_y$" hold, where $P|_x$ and $P|_y$ are the state predicates derived from P as discussed earlier. The implications are required because the static state information has to contain all relevant tuples in order to process a portion of Q by each processor independently.

Using the quantitative analysis that is presented earlier, the overhead of statically storing state information of a relation X at every partitioning boundaries is:

$$\sigma_q \cdot (n_{pi} - 1) \cdot \overline{T_{ls}}/TR_{ls}$$

where σ_q is the selectivity of the state predicate P_x, n_{pi} is the total number of partitions, $\overline{T_{ls}}$ is the average tuple lifespan and TR_{ls} is the lifespan of relation X.

In general, the more selective the state predicate is, the less space is required for storing state information statically. On the other hand, the class of queries

[14] Equivalently we can pick a query that subsumes a set of frequent queries.

[15] Note that if retroactive update is supported, it may be necessary to "refresh" the static state information as the updated tuple may span different partitioning intervals.

that can be processed in parallel without moving *additional* data (i.e., constructing state information dynamically) becomes smaller. Below, we will discuss some mechanisms in reducing the storage requirement of state information and their tradeoffs.

Ratio of Partition Boundaries & State Information The approach can be regarded as reducing the state information along the processor dimension. Specifically, only a subset of partitioning boundaries are selected for storing state information statically. That is, only a subset of processors will store static state information. For example, one can store state information at every other partitioning boundary. The tradeoffs include that not necessarily all processors have to participate in processing some queries. For example, suppose that relations are TS range-partitioned. Consider that a simple "as of" query such as finding the data value of a time-varying attribute of an object as of a particular time, e.g., $\sigma_{S=10}$ (X) as of t_i. In general, at least processors from p_1 to p_i would have to participate in processing the query. With state information stored at every other partitioning boundary, at most two processors (p_{i-1} and p_i) are required to perform the search.

Suppose that we statically store state information at one out of g partitioning boundaries. The overhead of storing state information statically in this approach is:

$$\sigma_q \cdot (n_{pi} - 1) \cdot \overline{T_{ls}}/(g \cdot TR_{ls})$$

Interestingly enough, one can view this approach from a different direction. In the original scheme, each processor stores a relation fragment and some state information. In the above approach, a portion of the relation fragment is statically moved and stored at another processor. From this point of view, the query processing workload is spread among the processors.

Partitioning Boundaries Another alternative is to select partitioning boundaries such that fewer tuples span the boundaries. For example, one can choose a time point t' as the partitioning boundary where there are many new tuples inserted right after time t'. If we had chosen a time point after t', many tuples (which span t') may have to be included as state information.

One can utilize this idea as suggested in Section 5.4. Consider two processors p_{i-1} and p_i which store partitions $[t_{i-1}, t_i)$ and $[t_i, t_{i+1})$ respectively. Instead of storing qualified tuples that span t_i as state information at p_i, we choose a time point t' where $t_{i-1}<t'\leq t_i$, and store only qualified tuples that span both t' and t_i. To process queries in parallel as before, processor p_i has to read from the partition $[t_{i-1}, t_i)$ from p_{i-1} but only the portion (t', t_i). In this scheme, processor p_{i-1} would send qualified tuples to p_i that belong to the partition (t', t_i) and span t_i.

The quantitative analysis is similar to the analysis presented in Section 5.4. Suppose that the distance between t' and t_i is $\bar{t}$ for all partitioning boundaries t_i, and that all tuple lifespans are independent and exponentially distributed with mean $\overline{T_{ls}}$. The overhead of storing state information statically at all partitions is:

$$P[y>\bar{t}\,] \cdot \sigma_q \cdot (n_{pi} - 1) \cdot \overline{T_{ls}}/TR_{ls}$$

where y is a random variable representing the tuple lifespan and $P[y>\bar{t}\,]$ is the probability that a tuple would span both time points t^- and t. The probability $P[y>\bar{t}\,]$ equals:

$$1 - P[0<y\leq\bar{t}\,] = 1 - \int_0^{\bar{t}} \frac{1}{\overline{T_{ls}}} \cdot e^{y/\overline{T_{ls}}}\, dy$$
$$= e^{-\bar{t}/\overline{T_{ls}}}$$

6.3.3 Late Updates

Although we assume that time points are monotonically increasing, update times are not necessarily so[16]. This leads to a problem if tuples are statically replicated as state information. Consider the TS range-partitioning scheme and a current tuple $r<s,v,t_2,now>$ stored at processor p_2. The issue is: since the TE timestamp of tuple r can be set to any time point between $[t_2+1,now)$, should tuple r be statically replicated in all processors from p_3 to $p_{n_{pi}}$ as the state information?

There are several ways to tackle this problem. A solution is to assume that we are dealing with transaction times and to enforce that database updates are

[16] Interesting enough, the Time Index proposed in [Elm90] did not address this issue. It appears that update times are implicitly assumed to be monotonically increasing.

performed in increasing order of transaction times. Ensuring the monotonicity can be readily achieved using a system clock.

Another straightforward solution is this. The tuple *r* is statically replicated in all other processors as state information. When the TE value of tuple r is updated to a specific value (say at t_j), the duplicates at processors whose partitioning boundary is greater than t_j are removed from the state information while the duplicates at processors whose partition boundary is smaller than or equal to t_j will be updated accordingly. This approach is optimistic in the sense that if most updates occur at a time greater than the last partitioning boundary ($t_{n_{pi}}$), not many duplicates will be removed. If we assume that "*now*" is the largest current time point, "*now*" would be greater than the latest partitioning boundary $t_{n_{pi}}$ and thus no duplicate will be removed.

An alternative approach is to specify that all current tuples (i.e., TE value is *now*) are not replicated; only history tuples (i.e., TE<*now*) are replicated as state information. In the next section, we further explore this alternative further, keeping in mind that current tuples are more frequently accessed via surrogate or attribute values rather than timestamp values.

6.3.4 Current Tuples and History Tuples

In the previous sections, we focus on processing of complex temporal pattern queries and snapshot queries. Although one should not ignore these queries, the most frequently accessed tuples may be predominantly the current tuples especially in conventional business-oriented applications. This type of access pattern may appear often for several reasons. First, many users may be interested in only current tuples and might view the temporal database as if it were a "static" current database. Moreover, updating the value of a time-varying attribute will modify the current tuple by setting its TE value ("*now*") to a specific time point. One of the characteristics of the access pattern in business-oriented applications is that current tuples are often accessed via a given surrogate (or key) value or a time-varying attribute value, e.g., accessing department records by name or department number. This suggests that a different fragmentation strategy for current tuples (instead of range-partitioning on timestamps) may provide more efficient access.

Hybrid Fragmentation Scheme To support efficient access to current tuples based on non-time attributes and to facilitate temporal query processing at the same time, we propose the following hybrid scheme:

- Current tuples are distributed among the processors using any fragmentation method. For the sake of explanation, we use hashing in our discussion.
- History tuples are range-partitioned on a timestamp (TS or TE) as proposed before.

That is, temporal relations are partitioned into two *logical* disjoint fragments: "current" and "history" fragments. The idea of this hybrid scheme is rather simple: as the temporal database is being updated, history tuples are "migrated" to a particular processor based on their timestamp values.

It should be emphasized that the two logical fragments (current and history fragments) can be stored in different file structures, although we might want to treat them as a single logical entity for discussion purposes. Such separation enables us to access the current fragment more efficiently. For example, one can create indices on the current fragment as in conventional databases. One can also avoid storing the special markers "*now*" in current tuples as they are really redundant in the current fragment. Nonetheless, for a temporal relation R that is TS range-partitioned, processor p_i logically keeps tuples specified as:

$$\begin{aligned} R_i = \{\, r \mid r \in R \wedge (\, (\, r.\mathrm{TE} = \mathit{now} \wedge h(r) = i\,) \vee \\ (\, r.\mathrm{TE} \neq \mathit{now} \wedge r.\mathrm{TS}\ \textsf{between}\ [t_i, t_{i+1})\,)\,)\,\} \end{aligned}$$

where $h(r)$ is a hash function to be applied on attribute(s) of R such as the surrogate.

Note that since temporal relations are logically partitioned into current and history fragments, the append-only update model that is presented at the beginning of this dissertation has to be slightly adjusted to incorporate the fact that updating a current tuple will generally migrate the corresponding history tuple between processors.

State Information & Temporal Query Processing The parallel processing strategies presented earlier basically remain unchanged in the hybrid scheme.

The state information is dynamically constructed using both current and history fragments. Since the current fragment is neither TS nor TE range-partitioned, the state information for partition $[t_i, t_{i+1})$ contains the following current tuples:

$$\{ r \mid r \in R \wedge t_i \leq r.TE \wedge r.TS \leq t_{i+1} \wedge P_r(r) \}$$

As in the case of TS or TE range-partitioning schemes, history tuples are also replicated as state information as before. After the replication phase, each processor can individually process its local history fragments and the state information.

For state information to be stored statically, we adopt a somewhat different strategy in that only qualified history tuples (i.e., no current tuples) are replicated and stored. For example, for the relation R that is TS range-partitioned, processor p_i statically keeps tuples specified as:

$$\begin{aligned} R_i = \{ r \mid r \in R \wedge (& (r.TE = now \wedge h(r) = i) \vee \\ & (r.TE \neq now \wedge ((r.TS \ \textbf{between} \ [t_i, t_{i+1})) \vee \\ & (r.TS < t_i \wedge t_i \leq r.TE \wedge P_r(r)))) \} \end{aligned}$$

where P_r is the predicate for constructing the static state information.

A point to note regarding the redundant state information property that is discussed earlier. For the hybrid fragmentation scheme, the property does not hold because the current fragment is not range-partitioned based on the TS or TE timestamp. However, if only the history fragment is involved in the user query, one can still eliminate the construction phase of an operand that has the asymmetry property. A condition in which only history fragment of a relation R_k is involved is given by:

$$P \Rightarrow R_k.TE \neq now$$

where P is the user query qualification. If we assume that *now* is the largest current time point, the above condition is equivalent to "$P \Rightarrow R_k.TE < now$".

6.3.5 ¬1NF temporal relations

There are several common ways to model temporal data. The data model that is presented earlier uses a pair of timestamps to represent the tuple lifespan and thus can be regarded as *tuple-versioning* [Ahn86] or *first temporal normal form* (1TNF) [Seg88]. Another approach is *attribute versioning* [Ahn86], including the

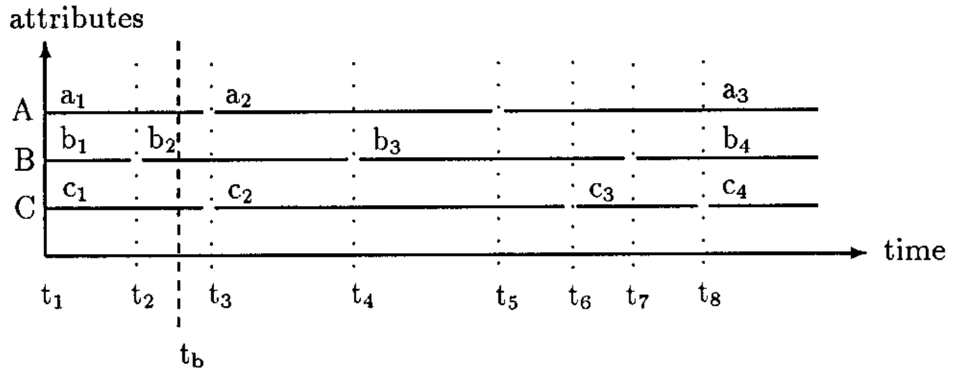


Figure 6.11: A temporal tuple (s_1) with multiple time-varying attributes

nested or *non-first normal form* (¬1NF) temporal data models [Gad88a, Tan89]. In the attribute versioning approach, a temporal relation may have more than one time-varying attribute; each attribute value is tagged with a pair of timestamps representing its lifespan. Consider a relation with three time-varying attributes A, B and C:

$$R(S,\ A,TS_A,TE_A,\ B,TS_B,TE_B,\ C,TS_C,TE_C)$$

Typically, a tuple has a surrogate value and several timestamped data values for each attribute. For this reason, the tuple is in ¬1NF. For example, Figure 6.11 shows the attribute values of a tuple (s_1) over time. The corresponding ¬1NF tuple is shown in Figure 6.12 and its normalized version is shown in Figure 6.13. For ¬1NF relations, the database update model remains essentially unchanged — the difference is that the append-only update policy is applied on the attribute level and thus no additional ¬1NF tuple is inserted when an attribute value is updated. In other words, the size of a ¬1NF tuple grows as more history attribute values are appended to it. It should be noted that a temporal relation in ¬1NF (e.g., Figure 6.12) is conceptually equivalent to a materialized join on surrogate value using the corresponding normalized relations (e.g., Figure 6.13), although a different file structure may be used for storing ¬1NF tuples.

Range-partitioning ¬1NF relations on a time attribute is rather straightforward as before. Let us consider the hybrid scheme on TS range-partitioning:

S	A $[TS_A, TE_A)$	B $[TS_B, TE_B)$	C $[TS_C, TE_C)$
s_1	a_1 $[t_1,t_3)$	b_1 $[t_1,t_2)$	c_1 $[t_1,t_3)$
	a_2 $[t_3,t_5)$	b_2 $[t_2,t_4)$	c_2 $[t_3,t_6)$
	a_3 $[t_5,now)$	b_3 $[t_4,t_7)$	c_3 $[t_6,t_8)$
		b_4 $[t_7,now)$	c_4 $[t_8,now)$

Figure 6.12: A ¬1NF tuple for s_1

S	A	TS_A	TE_A
s_1	a_1	t_1	t_3
s_1	a_2	t_3	t_5
s_1	a_3	t_5	*now*

S	B	TS_B	TE_B
s_1	b_1	t_1	t_2
s_1	b_2	t_2	t_4
s_1	b_3	t_4	t_7
s_1	b_4	t_7	*now*

S	C	TS_C	TE_C
s_1	c_1	t_1	t_3
s_1	c_2	t_3	t_6
s_1	c_3	t_6	t_8
s_1	c_4	t_8	*now*

Figure 6.13: Normalization of tuple s_1

- The current fragment of s_1 is represented by the following tuple, and is stored at a processor determined by a fragmentation method such as hashing:

s_1	$a_3\ [t_5, \mathit{now})$	$b_4\ [t_7, \mathit{now})$	$c_4\ [t_8, \mathit{now})$

  Note that the start times of current attribute values are different.

- The history fragment is range-partitioned based on the TS timestamp. To explain the idea, we consider a partitioning boundary t_b as shown in Figure 6.11. History attribute values which start prior to t_b are stored at one processor, say p_1, while those after t_b are stored at another processor, say p_2. For tuple s_1, processor p_1 stores the following portion:

s_1	$a_1\ [t_1, t_3)$	$b_1\ [t_1, t_2)$ $b_2\ [t_2, t_4)$	$c_1\ [t_1, t_3)$

  while p_2 stores:

s_1	$a_2\ [t_3, t_5)$	$b_3\ [t_4, t_7)$	$c_2\ [t_3, t_6)$ $c_3\ [t_6, t_8)$

- State information can be constructed and stored statically. Consider the most general case in which the state predicate to construct the static state information is "true". The portion of tuple s_1 stored in processor p_2 becomes:

s_1	$a_1\ [t_1, t_3)$ $a_2\ [t_3, t_5)$	$b_2\ [t_2, t_4)$ $b_3\ [t_4, t_7)$	$c_1\ [t_1, t_3)$ $c_2\ [t_3, t_6)$ $c_3\ [t_6, t_8)$

- Updating an attribute value may "migrate" the history attribute value. For example, suppose the attribute A of tuple s_1 is updated to a_4 at time t_9 (which is later than t_8). The history value a_3 is then moved to processor p_2 which would store (without state information):

s_1	$a_2\ [t_3, t_5)$ $a_3\ [t_5, t_9)$	$b_3\ [t_4, t_7)$	$c_2\ [t_6, t_8)$ $c_3\ [t_6, t_8)$

6.3.6 Temporal Aggregate Functions

The notion of state information is not limited to qualified tuples that span partitioning boundaries, and can be further generalized in the context of stream processing. Recall that the state information of a stream processor at a particular time t' represents a summary of the history of a computation on the portion of data streams that have been read before t'. Generally speaking, it may be very difficult to characterize the state information (and therefore its storage requirement) for an arbitrary computation. However, the notion of state information can be easily defined for aggregate functions.

Suppose that we ask an aggregate query: *find the weekly sales volume from the daily sales records*[17]. We further suppose that the data is range-partitioned on a yearly basis among processors, e.g., processors p_{i-1} and p_i store the partitions for the years of t_{i-1} and t_i respectively. That is, the temporal data is not time-interval tuples. The state information at processor p_i can be defined as:

> tuples whose timestamp values are during the last week of the year of t_{i-1}.

To process the aggregate query in parallel, each processor p_i has to send (at most) tuples in the last week of its local fragment to its "successor" processor. For example, p_i sends copies of some tuples in its local fragment (i.e., year of t_i) to p_{i+1} as state information. Note that the number of tuples in the state information for this aggregate query is bounded — at most one week of daily sales records.

6.4 Other Parallel Temporal Join Strategies

The strategies that are presented earlier may not be applicable in some situations, e.g., TSJ_2 queries. In this section, we discuss several alternative parallel strategies.

[17] Several temporal aggregation operators have been proposed and defined in [Sno86, Seg87]. An example taken from [Seg87] is: "Get a series of 7-day moving averages of book sales."

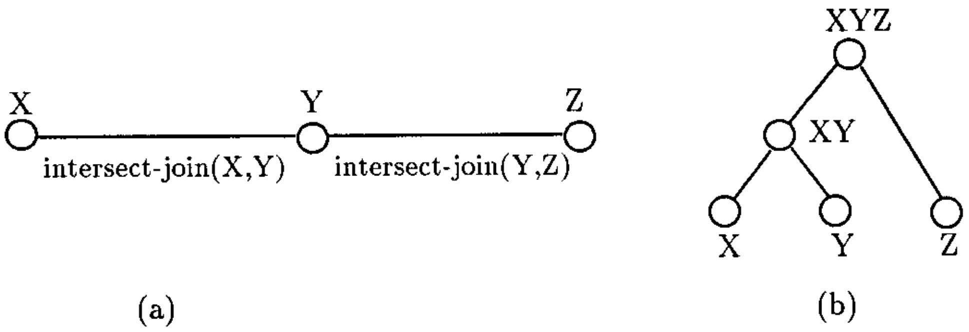


Figure 6.14: Join graph for the TSJ_2 query in Example 6.5

6.4.1 TSJ_2 Queries

In the previous sections, we discuss the parallel processing strategies that are suitable for TSJ_1 queries only. Here we explain why these strategies cannot be utilized for a more generally class of join queries — TSJ_2, and suggest some alternatives in processing these queries. Recall that TSJ_2 are join queries whose join conditions do not require all participating tuples to share a common time point (but the participating tuples must overlap).

Example 6.5 Given temporal relations X, Y and Z. The following query belongs to TSJ_2 (but not TSJ_1):

$$\sigma_{\text{intersect-join(X,Y)} \wedge \text{intersect-join(Y,Z)}} (X,Y,Z)$$

and its join graph is shown in Figure 6.14(a). □

Consider Example 6.5. As in the case for TSJ_1, we assume relations are homogeneously range-partitioned on time attributes. For a tuple triplet $< x, y, z >$ that satisfies the join condition, the tuples x and z may not overlap with each other and thus may actually reside on different processors. In other words, even if sufficient state information for *all* operand relations have been constructed as before, the query may not be processed in parallel without moving additional data. The essential difference here is that the state information of temporary tables may have to be constructed dynamically during the join sequence while it is not required for TSJ_1 queries.

Processing TSJ_2 queries can be somewhat more complicated than TSJ_1 queries in general. For example, we can first construct a join sequence for pairwise join execution using the join graph built from the join query qualification. In Example 6.5, it is "X-Y-Z". Suppose the join sequence is to join relations X and Y first, and then join the result with relation Z, as illustrated in Figure 6.14(b)[18]:

Replication Phase I Construct the state information for relations X, Y and Z, denoted as $s_{t_i}(X)$, $s_{t_i}(Y)$ and $s_{t_i}(Z)$ respectively, at every partitioning boundary t_i, as described before.

X-Y Join Execute the join between X and Y in parallel, the result being a temporary table (denoted as XY) which is also fragmented among processors. The fragment of XY stored at p_i is:

$$XY_i = \sigma_{Pxy}(X_i, s_{t_i}(Y)) \cup \sigma_{Pxy}(Y_i, s_{t_i}(X)) \cup \sigma_{Pxy}(X_i, Y_i)$$

where X_i (and Y_i respectively) is a relation fragment of X (and Y respectively) stored at processor p_i, and Pxy is "intersect-join(X,Y)".

Replication Phase II Construct the state information for relation XY at every partition. That is, A tuple pair $< x, y >$ in XY is copied to processor p_i as state information, denoted as $s_{t_i}(XY)$, if its y component spans t_i.

XY-Z Join The final query response becomes joining the Z fragment with XY fragment and $s_{t_i}(XY)$. That is,

$$Q = \bigcup_{1 \le i \le n} \{\sigma_{Pyz}(XY_i, s_{t_i}(Z)) \cup \sigma_{Pyz}(Z_i, s_{t_i}(XY)) \cup \sigma_{Pyz}(XY_i, Z_i)\}$$

where Z_i is a relation fragment of Z stored at processor p_i, and Pyz is "intersect-join(Y,Z)".

[18] Obviously, there is a tradeoff in choosing which join to be processed first. For example, joining Y and Z may produce a smaller temporary relation than joining X and Y, and therefore the overall cost may be cheaper if we join Y and Z first.

6.4.2 Sequential/Pipelining

This strategy assumes that relations are homogeneously TS range-partitioned. In conventional databases, "pipelining" often refers to the paradigm in which data "flow" through relational operators such as join and select without being stored in temporary files. The sequential/pipelining strategy here refers to the dataflow among processors and to the fact that state information is sent from one processor to another sequentially. The approach stems from the observation that the processor p_1, which stores tuples in the first partition $[t_1,t_2)$, can immediately proceed to execute the join since we assume that the state information at t_1 (equal to 0) is empty. While the local join in processor p_1 proceeds, qualified tuples that span the next partitioning boundary t_2 are copied to processor p_2. Processor p_2 can start its execution at any time but it can not finish the execution until it has received all the necessary state information from p_1. This execution mechanism continues for p_2 until $p_{n_{pi}}$ which stores the last partition $[t_{n_{pi}},now)$.

This sequential/pipelining scheme can significantly reduce the interconnection network traffic congestion due to the simultaneous tuple shuffling among all processors. The initial query response is faster as the first processor can start the execution without delay. However, processing the query takes longer as it has to go through from a processor to another, and the total query response time may be too long that pipelining becomes unacceptable especially when operand relations are range-partitioned into many fragments.

6.4.3 Semijoin and Join

This method can handle both homogeneous as well as heterogeneous range-partitioning. The semijoin is actually a pre-processing mechanism that can also reduce the number of tuples copied between processors.

Recall that in an earlier section, we discuss alternatives in reducing the amount of state information using four statistics: the maximum and minimum of the TS and TE timestamp values of a relation fragment X_i. The idea is that instead of keeping these four statistics, each processor p_i scans the operand fragment X_i and determine its *exact* fragment lifespan which is obtained by "concatenating" the lifespans of overlapping tuples in X_i. Note that the fragment lifespan may consist of more than one interval[19]. Using the fragment lifespan of X_i in processor p_i, each processor can determine which Y tuples should be copied to p_i for executing the join, for example, sending Y tuples which overlap with the fragment lifespan of X_i. Readers may note that only relation X is required to be range-partitioned on a timestamp; other fragmentation strategies (such as hashing) can be used on relation Y.

One can compare this approach with the alternative that uses the four statistics discussed earlier. The above approach has to pay the overhead in determining fragment lifespans (e.g., scanning fragments once). However, fewer tuples may be sent between processors especially when the fragment lifespans are several disjoint intervals. That is, there are "gaps" in the fragment lifespan that can be used to avoid sending redundant tuples as state information.

6.5 Previous Work and Conclusions

The parallel processing schemes that we present in this chapter is a substantial extension of the work on generalized data stream indexing in Chapter 5 — the notion of checkpointing the execution state of a query appears in both chapters. In Chapter 5, we propose an indexing technique based on periodically checkpointing on data streams which are sorted on the timestamp values. Checkpoints are statically stored and can be indexed on checkpoint times. In this chapter, we apply the idea of checkpointing in parallel database machines.

[Kar90] is apparently the first publication that appears to support temporal features in multiprocessor database machines. The paper, however, only discusses a front-end syntactic translator for a relational database system regardless of whether or not the database system is residing on a multiprocessor database machine. Moreover, there is no discussion on query processing and optimization as well as fragmentation strategies.

A partitioned storage for temporal databases is proposed in [Ahn88]. The idea is to split the storage structure into the *history store* and the *current store*, and is similar to the hybrid range-partitioning scheme proposed in this chapter.

[19] If the query qualification has a comparison predicate on relation X (e.g., "X.U=u") and an index on that particular attribute exists, one can use the index to retrieve qualifying tuples and determine the "fragment lifespan".

The current store contains current versions and perhaps some history versions of temporal records while the history store contains only history versions. The major issues involved are 1) mapping temporal queries into queries on two storage structures, and 2) update procedures. Their emphasis is on select queries and equi-joins. The idea of archiving a portion of history tuples into optical disks also appears in [Sto87, Kol90].

In [DeW91] a "partitioned band" join algorithm is proposed to evaluate the so-called "band join"[20]:

> A "band join" between relations R and S on attributes R.A and S.B is a join in which the join condition is "$R.A - c_1 \leq S.B \leq R.A + c_2$", where c_1 and c_2 are non-negative constants.

In the band join algorithm, ranges of the operand relations R_i and S_i, where $i \in \{1, \cdots, n\}$, are found such that (1) $R = \bigcup_i R_i$ and $S = \bigcup_i S_i$, and (2) for every tuple r in R_i, it is required that all tuples of S that join with r appear in S_i. The complete join is formed by joining R_i and S_i for each range ($i=1, \cdots, n$) and merging the result. With the assumption that the width of a "band" (i.e., $c_1 + c_2$) is small, the major concern in [DeW91] is to choose the range sizes such that each of the R_i fits entirely into the buffer pool. For the parallel version of the band join algorithm, each join between ranges R_i and S_i can be performed by a separate processor.

One can process the above band join using our strategies as follows. Suppose both relations R and S are range-partitioned based on the join attribute (R.A and S.B) using the same partition function (this assumption is easily relaxed and would just result in greater data movement as discussed below). We further suppose that a partitioning interval $[v_i, v_{i+1})$ is assigned to a processor p_i, i.e., a tuple $r \in R$ (similarly for tuples in S) is stored at p_i if its join attribute value falls into this interval. The replication phase then involves copying tuples $s \in S$ to p_i if the value of $s.B$ falls into the interval $[v_i - c_1, v_i)$ or $[v_{i+1}, v_{i+1} + c_2)$. When both c_1 and c_2 are small, tuples from only processors p_{i-1} and p_{i+1} are replicated at processor p_i. After the replication process, the join can be processed as the merging of the results of the parallel local joins. If say R is partitioned on the

[20] One can think of it as a "fuzzy" equi-join.

join attribute but S is not, then the same strategy works except that the tuples replicated on processor p_i may come from all other processors.

To recap, we discuss parallel query processing strategies for complex temporal join queries and snapshot queries, and show that the strategies are sound for TSJ_1 queries. A number of optimization alternatives have been addressed. We have also discussed several data fragmentation strategies for temporal data.

CHAPTER 7

Conclusions and Future Work

7.1 Contributions

The focus of this work has been on processing and optimization strategies for temporal queries. In the following, we summarize our major contributions to this area.

- We show that most temporal operators (except for the time-union operator) are equivalent to relational expressions, and thus establish the argument that the major distinction between temporal DBMS and relational DBMS is in the area of query processing and optimization. In fact, this area is seldom explored until recently [Seg89, Leu90, Gun91].

- We propose stream processing techniques for processing temporal join and semijoin operations. Stream processing algorithms, which takes advantage of the ordering of input streams, can be very efficient. We also discuss the trade-offs between the input sort orderings, minimal size of workspace where state information is kept and the number of scans of input streams. Note that these trade-offs are seldom addressed in the existing literature. One of the major contributions is that the optimal sort orderings may depend on the statistics of data instances as well as the operator itself. For example, it may be more efficient if an input stream is sorted on TS while the other input stream is sorted on TE (such as for contain-join).

- We propose generalized data stream indexing techniques that can be used to efficiently process a subclass of complex joins qualified with a snapshot operator, especially when the query-specific time interval (e.g., in as of queries) is relatively short compared with the lifespans of data streams. The data stream indices become very attractive especially when the storage space required for the indices is relatively small compared with conventional indices such as B$^+$tree. One of the major contributions is the set of storage reduction schemes where the amount of state information that we store as checkpoints can be reduced, and the fact that there is a limit on the gain due to this optimization scheme.

- We propose parallel processing algorithms and optimization strategies for a subclass of temporal joins. The parallel algorithms are based on partitioning the relations along a time dimension. We also consider a number of alternatives in reducing the number of tuples to be replicated across processors. A quantitative analysis is provided to estimate the number of tuples to be replicated, and based on this quantity, one can determine when our proposed strategies perform better than conventional methods.

- We classify temporal join queries into TSJ$_1$ and TSJ$_2$. This classification has two major consequences. First, it helps us understand the applicability of various algorithms and strategies on processing these types of queries. A by-product is that it identifies the limitations of our generalized data stream indexing techniques and the parallel processing algorithms in that they are not directly applicable for TSJ$_2$ queries. Second, although we concentrate most on TSJ$_1$ queries, the algorithms and strategies presented can form the foundation for processing TSJ$_2$ queries.

- This research work opens up many avenues for future work in temporal query processing and optimization that have been largely ignored in the past. In the following section, we point out several directions of the future research.

7.2 Future Work

There are many research directions which require further investigation. They include:

- statistical information gathering,
- generalized stream processing algorithms,
- global optimization problem,

- semantic query optimization,
- parallel processing strategies for TSJ_2 queries and queries whose operand relations are heterogeneously TS or TE range-partitioned.

Statistical Information Gathering Statistical information about databases is known to be important in query optimization. For temporal databases, it appears to be more critical. In addition to conventional statistical information such as relation size and image size of indices, estimating the amount of local workspace becomes necessary. There is also a question of how this information can be obtained efficiently and summarized in a suitable form for the optimizer.

Generalized Stream Processing Algorithms In Chapter 4, we assume that the workspace is large enough for holding all the necessary tuples so that the join operation can be performed by scanning input relations once. More general algorithms are required to handle the situation when overflow occurs. Assuming that the query optimizer requests a workspace of certain size for a particular operation before a stream processing algorithm starts, there are several alternatives to handle the "overflow" problem:

- The workspace is managed by a virtual memory manager, i.e., the workspace is mapped onto a swap space. This simple solution may have a severe performance penalty as there is no control over the paging activities.
- When overflow occurs, the join operation can be "continued" using 3 separate joins. Let us outline the approach here. Suppose we join data streams X and Y. We denote the portion of data stream X (respectively Y) that have not been read as the "remainder" of X (respectively Y). The first two joins are to join the state information of X with the remainder of Y, and vice versa. This can be done via scanning the reminders of both data streams only once. The last join is to join the remainders of both data streams using a nested-loop join. Note that estimating the expected performance of this kind of generalized stream processing algorithms is important and worth the investigation.

Global Optimization Problem One of the most important area for future work is the "global" optimization problem which can be stated as follows.

> Generally a query optimizer is given the following information:
>
> - a list of available indices,
> - a list of available join strategies,
> - the data statistics,
> - the sort ordering of input operands,
> - the available workspace, and
> - the cost model.
>
> For a given user TSJ query in the form of $\sigma_P(R_1, \cdots, R_m)$, where R_i's ($1 \leq i \leq m$) are temporal relations and P is a query qualification (i.e., comparison and join predicates), the query optimizer generates a query plan which is a sequence of operations (such as sorting the input data and performing a selected join strategy) which includes determining of the join ordering. The global optimization problem is to choose a plan with the cheapest cost.

We note that most of the research work in temporal databases to date has only considered storage structures, query processing algorithms for simple temporal queries (such as select and join), and indexing methods. The point here is that in addition to the new strategies for *individual join operation*, we should also consider the global optimization problem.

Semantic Query Optimization In Chapter 4, we note that time is rich in semantics, and one can exploit semantic query optimization techniques in generating a better query plan. Its use will be more crucial when the global query optimization problem is tackled, and it should be addressed in the future.

Parallel Query Processing Strategies In Chapter 6 we concentrate on TSJ_1 queries (i.e., all participating tuples that satisfy the join condition must share a common time point). A natural extension is investigate the parallel query processing strategies for TSJ_1 queries whose operand relations that are heterogeneously range-partitioned, and for TSJ_2 queries. It appears that our parallel join strategies presented here can be easily adopted to process TSJ_2 queries: the join sequence can be obtained by a graph reduction algorithm on the join graph constructed using Algorithm 3.1. These areas need to be investigated further.

References

[Abe85] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[Ahn86] I. Ahn. Towards an Implementation of Database Management Systems with Temporal Support. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 374–381, February 1986.

[Ahn88] I. Ahn and R. Snodgrass. Partitioned Storage for Temporal Databases. *Information Systems*, 13(4):369–391, 1988.

[All83] J. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[Ban87] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 311–322, May 1987.

[Bar85] F. Barbic and B. Pernici. Time Modeling in Office Information Systems. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 51–62, May 1985.

[Bat88] D.S. Batory, T.Y. Leung, and T.E. Wise. Implementation Concepts for an Extensible Data Model and Data Language. *ACM Trans. on Database Systems*, 13(3):231–262, September 1988.

[Ben79] J. Bentley. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Trans. on Computers*, 28(9):643–647, September 1979.

[Ben80] J. Bentley and D. Wood. An Optimal Worst Case Algorithm for Reporting Intersections of rectangles. *IEEE Trans. on Computers*, 29(7):571–578, July 1980.

[Ben82] J. Ben-Zvi. *The Time Relational Model.* PhD thesis, University of California at Los Angeles, 1982. Department of Computer Science.

[Bor82] H. Boral and D. DeWitt. Applying Data Flow Techniques to Data Base Machines. *IEEE Computer*, 15(8):57–63, August 1982.

[Cha88] S. Chaudhuri. Temporal Relationships in Databases. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 160–170, 1988.

[Chak84] U.S. Chakravarthy, D.H. Fishman, and J. Minker. Semantic Query Optimization in Expert System and Database Systems. In *Expert Database Systems*, pages 326–341, 1984.

[Cho90] J. Chomicki. Polynomial Time Query Processing in Temporal Deductive Databases. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 379–391, 1990.

[Cli83] J. Clifford and D.S. Warren. A Model for Historical Databases. *ACM Trans. on Database Systems*, 8(2):214–254, June 1983.

[Cli85] J. Clifford and A. Tansel. On an Algebra for Historical Relational Databases: Two Views. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 247–265, May 1985.

[Cli87] J. Clifford and A. Croker. The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 528–537, February 1987.

[Cod79] E.F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Trans. on Database Systems*, 4(4):397–434, December 1979.

[DeW90] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Trans. on Knowledge and Data Engineering*, March 1990.

[DeW91] D. DeWitt, J. Naughton, and D. Schneider. An Evaluation of Non-Equijoin Algorithms. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 443–452, 1991.

[Dut89] S. Dutta. Generalized Events in Temporal Databases. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 118–125, 1989.

[Elm90] R. Elmasri, G. Wuu, and Y. Kim. The Time Index: An Access Structure for Temporal Data. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 1–12, 1990.

[Elm91] R. Elmasri, Y. Kim, and G. Wuu. Efficient Implementation Techniques for the Time Index. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 102–111, 1991.

[Gad88] S. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM Trans. on Database Systems*, 13(4):418–448, December 1988.

[Gad88a] S. Gadia and C. Yeung. A Generalized Model for a Relational Temporal Database. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 390–397, June 1988.

[Gha90] S. Ghandeharizadeh and D. DeWitt. Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 481–492, 1990.

[Gun89] H. Gunadhi and A. Segev. Efficient Indexing Methods for Temporal Relations. Technical Report LBL-28798, University of California at Berkeley, Lawrence Berkeley Laboratory, 1989.

[Gun91] H. Gunadhi and A. Segev. Query Processing Algorithms for Temporal Intersection Joins. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 336–344, 1991.

[Gut84] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, June 1984.

[Jar84] M. Jarke. External Semantic Query Simplification: A Graph-theoretic Approach and its Implementation in Prolog. In *Expert Database Systems*, pages 467–482, 1984.

[Jen91] C. Jensen and R. Snodgrass. Temporal Specialization and Generalization. Technical Report TR 91-25, University of Arizona at Tucson, Computer Science Department, 1991.

[Kab90] F. Kabanza, J. Stevenne, and P. Wolper. Handling Infinite Temporal Data. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 392–403, 1990.

[Kaf90] W. Kafer, N. Ritter, and H. Schoning. Support for Temporal Data by Complex Objects. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 24–35, 1990.

[Kar90] S. Karimi, M. Bassiouni, and A. Orooji. Supporting Temporal Capabilities in a Multi-computer Database System. In *Proc. of the Int.*

Conf. on Databases, Parallel Architectures, and their Applications, pages 20–26, March 1990.

[Kim88] W. Kim and H. Chou. Versions of Schema for Object-Oriented Databases. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 148–159, 1988.

[Kin81] J. King. QUIST: A System for Semantic Query Optimization in Relational Databases. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 510–517, 1981.

[Kol89] C. Kolovson and M. Stonebraker. Indexing Techniques for Historical Databases. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 127–137, February 1989.

[Kol90] C. Kolovson. *Indexing Techniques for Multi-Dimensional Spatial Data and Historical Data in Database Management Systems.* PhD thesis, University of California at Berkeley, November 1990. Memorandum No. UCB/ERL M90/105.

[Kol91] C. Kolovson and M. Stonebraker. Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 138–147, 1991.

[Kow86] R. Kowalski and M. Sergot. A Logic-based Calculus of Events. *New Generation Computing*, 4(1):67–95, 1986.

[Lam83] L. Lamport. What good is Temporal Logic? In *Information Processing, IFIP*, pages 657–668, 1983.

[Leu90] T.Y. Leung and R.R. Muntz. Query Processing for Temporal Databases. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 200–207, 1990.

[Leu91] T.Y. Leung and R.R. Muntz. Stream Processing: Temporal Query Processing and Optimization. Technical Report CSD-910079, University of California, Los Angeles, Dept. of Computer Science, November 1991.

[Lin87] J. Lingat, P. Nobecourt, and C. Rolland. Behaviour Management in Database Applications. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 185–196, 1987.

[Lip87] U.W. Lipeck and G. Saake. Monitoring Dynamic Integrity Constraints Based on Temporal Logic. *Information Systems*, 12(3):255–269, 1987.

[Lit61] J. Little. A Proof of the Queueing Formula $L = \lambda W$. *Operational Research*, 9, 1961.

[Lom89] D. Lomet and B. Salzberg. Access Methods for Multiversion Data. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 315–324, 1989.

[Lum84] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, and J. Woodfill. Designing DBMS support for the Temporal Dimension. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 115–130, June 1984.

[McK91] Edwin McKenzie and R. Snodgrass. Evaluation of Relational Algebras incorporating the Time Dimension in Databases. *ACM Computing Surveys*, 23(4):501–544, 1991.

[Naq89] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, New York, 1989.

[Nar89] S. Narain and J. Rothenberg. A Logic for Simulating Dynamic Systems. In *Proc. of Winter Simulation Conference*, Washington, D.C., 1989.

[Nav88] S.B. Navathe and R. Ahmed. Temporal Aspects of Version Management. *Database Engineering*, 7(4):34–37, December 1988.

[Ngu89] A.H.H. Ngu. Transaction Modelling. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 234–241, February 1989.

[Nie84] J. Nievergelt, H. Hinterberger, and Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. on Database Systems*, 9(1):38–71, March 1984.

[Obe87] A. Oberweis and G. Lausen. On the Representation of Temporal Knowledge in Office Systems. In *Proceedings of the Conf. on Temporal Aspects in Information Systems*, pages 131–146, May 1987.

[Ore88] J. Orenstein. PROBE Spatial Data Modeling and Query Processing in an Image Database Application. *IEEE Trans. on Software Engineering*, 14(5):611–629, May 1988.

[Par89] D.S. Parker, R.R. Muntz, and H.L. Chau. The Tangram Stream Query Processing System. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 556–563, February 1989.

[Par90] D.S. Parker. Stream Data Analysis in Prolog. In Leon Sterling, editor, *The Practice of Prolog*. The MIT Press, Cambridge, MA, 1990.

[Pre85] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[Ros80] D. Rosenkrantz and H. Hunt. Processing Conjunctive Predicates and Queries. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 64–72, 1980.

[Rot87] D. Rotem and A. Segev. Physical Design of Temporal Databases. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 547–553, 1987.

[Seg87] A. Segev and A. Shoshani. Logical Modeling of Temporal Data. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 454–466, May 1987.

[Seg88] A. Segev and A. Shoshani. The Representation of a Temporal Data Model in the Relational Environment. In *Proc. of the Conf. on Statistical and Scientific Database Management*, pages 39–61, June 1988.

[Seg89] A. Segev and H. Gunadhi. Event-join Optimization in Temporal Relational Databases. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 205–215, August 1989.

[Sel79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, May 1979.

[She89] S.T. Shenoy and Z.M. Ozsoyoglu. Design and Implementation of a Semantic Query Optimizer. *IEEE Trans. on Knowledge and Data Engineering*, 1(3):344–361, September 1989.

[Sho86] A. Shoshani and K. Kawagoe. Temporal Data Management. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 79–88, 1986.

[Smi75] J. Smith and P. Chang. Optimizing the Performance of a Relational Algebra Database Interface. *Communications of the ACM*, 18(10):568–579, October 1975.

[Sno85] R. Snodgrass and I. Ahn. A Taxomony of Time in Databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 236–246, May 1985.

[Sno86] R. Snodgrass and S. Gomez. Aggregates in the Temporal Query Language TQuel. Technical Report 86-009, University of North Carolina, Chapel Hill, Dept. of Computer Science, March 1986.

[Sno87] R. Snodgrass. The Temporal Query Language TQuel. *ACM Trans. on Database Systems*, 12(2):247–298, June 1987.

[Soo91] M.D. Soo. Bibliography on Temporal Databases. *The ACM SIGMOD Record*, 20(1):14–23, March 1991.

[Sri88] S. Sripada. A Logical Framework for Temporal Deductive Databases. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 171–182, 1988.

[Ste86] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* The MIT Press, 1986.

[Sto86] M. Stonebraker. The case for shared nothing. *Database Engineering*, 9(1), 1986.

[Sto87] M. Stonebraker and L. Rowe. The POSTGRES Papers. Technical Report UCB/ERL M86/85, University of California at Berkeley, Electronic Research Laboratory, June 1987.

[Sto88] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. The Design of XPRS. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 318–330, 1988.

[Sto89] M. Stonebraker. The Case for Partial Indexes. *The ACM SIGMOD Record*, 18(4):4–11, December 1989.

[Sun89] X. Sun, N. Kamel, and L. Ni. Solving Implication Problems in Database Applications. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 185–192, June 1989.

[Tan89] A.U. Tansel and L. Garnett. Nested Historical Relations. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 284–294, June 1989.

[Ter85] Teradata Corporation. *DBC/1012 Database Computer System Manual Release 2.0*, November 1985. Document No. C10-0001-02.

[Tuz90] A. Tuzhilin and J. Clifford. A Temporal Relational Algebra as a Basis for Temporal Relational Completeness. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 13–23, 1990.

[Ull82] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, MD, second edition, 1982.

[Val87] P. Valduriez. Join Indices. *ACM Trans. on Database Systems*, 12(2):218–246, June 1987.