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

# STREAM PROCESSING: TEMPORAL QUERY PROCESSING AND OPTIMIZATION

T.-Y. C. Leung
R. R. Muntz

December 1991
CSD-910079

# Stream Processing: Temporal Query Processing and Optimization*

T.Y. Cliff Leung
Richard R. Muntz
University of California, Los Angeles

December 5, 1991

**Abstract**

In this paper, we present a stream processing approach to query processing and optimization in temporal databases. We first discuss the distinctions between timestamps which store relevant time information and ordinary integer-based attributes, and show that most of the temporal operators found in the literature are equivalent to traditional relational algebra. However, temporal data and queries have several intrinsic characteristics that can be exploited in query processing and optimization. Based on the stream processing paradigm, we discuss the processing strategies for various temporal operators. A summary of the current status in the area of temporal query processing and optimization, and some research issues that should be addressed in the future are discussed.

*This work was partially supported by a MICRO grant from the University of California and the Hughes Aircraft Company.

## 1 Introduction

Many real database applications intrinsically involve time-varying information. With the availability of cheap processing and storage units, there is a growing interest in temporal databases which store the evolving history of the "enterprise" of interest. A common approach to implementing a temporal DBMS can be described as follows:

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

Fundamentally there is no difference between a timestamp which stores relevant time information and an ordinary integer-based attribute. However, temporal data and queries provide several unique characteristics and challenges for query processing. We will argue that ignoring these characteristics can result in orders of magnitude poorer performance.

In this paper, we discuss a stream processing approach, which takes advantage of data ordering, for processing various temporal join and semijoin operations which are the most common and expensive computations in database systems. We note that temporal join and semijoin operators often contain a conjunction of several inequality predicates involving only timestamps. As temporal data often has certain implicit ordering by time, we will demonstrate that the stream processing approach is often the strategy of choice.

We also study the processing of the complex snapshot queries. That is, the query is restricted to tuples that are active as of a particular time or over a certain 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" operator. The strategy, which is based on the stream processing paradigm, is to provide an indexing mechanism such that tuples in the proximity of the query-specific time interval or time point can be retrieved efficiently.

The organization of this paper is as follows. In Section 2 we present the data model and the types of query that we consider. Section 3 is devoted to the discussion of the differences between timestamps and ordinary attributes. We show in Section 4 that all temporal operators (except the time-union operator) that appear in the literature can be translated into equivalent relational expressions. A stream processing approach is presented in Section 5, and the generalized data stream indexing technique is proposed in Section 6. In Section 7 we discuss query optimization issues involving the time-union operator which cannot be implemented using the five conventional relational operators. Section 8 contains a summary and a perspective on future research.

## 2 Data Model

In the temporal data model, time points are regarded as integers { 0, 1, $\cdots$, *now* } which are monotonically increasing and where *now* is a special marker that represents the current time. A time-interval temporal relation is denoted as X(S,V,TS,TE), where S is the surrogate, V is a time-varying attribute, and the interval [TS,TE) denotes the lifespan of a tuple. The TS and TE attributes are referred to as time attributes (or simply timestamps) while other attributes are referred to as non-time attributes. All relations are assumed to have a homogeneous lifespan — [0,*now*). We also assume that for each tuple, the TS value is always smaller than the TE value. That is, for each tuple $<s,v,ts,te>$, "$ts<te$" must hold. Using the taxonomy in [Sno87], the TS and TE attributes are called the *effective* timestamps as opposed to the *transaction* timestamps, and a database system which handles effective times is called a "historical database". 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.

We discuss the classification of several types of Temporal Select-Join (denoted TSJ) queries; each class has a restricted form of *query qualification* which is defined as a conjunction of a number of comparison predicates and/or join predicates. The classification allows us to study the difficulty and complexity of query processing and optimization for each class, and therefore helps us decide what strategies may be more suitable for a particular class of queries. The characterizations of these queries can be informally stated as follows:

**Disjoint Join** The join condition between two tuples does not require that their lifespans overlap, as illustrated in Figure 1(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 lifespans 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 1(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 1(c). However, the participating tuples that satisfy the join condition do not have to have 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 $TSJ_1$ queries also belong to $TSJ_2$.

Generally speaking, these queries can be difficult and expensive to process. Studying the characteristics for each query category in more detail, as we demonstrate in a later section, provides some

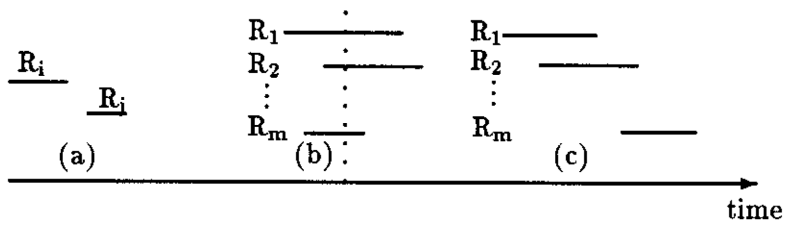

Figure 1: Classes of temporal joins

new alternatives in achieving more efficient query processing strategies. This is particularly true for "overlap joins" which will be the main focus in this paper.

We now precisely define several 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 1. Based on the join graph, we are able to formally define $TSJ_1$ and $TSJ_2$ join queries.

**Algorithm 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 $R_j$ ($i \neq j$) into G if the following condition is satisfied:

$$P(R_1,\cdots,R_m) \Rightarrow R_i.TS \leq R_j.TE \wedge R_j.TS \leq R_i.TE^{1}$$

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.

**Definition: $\mathbf{TSJ_2}$** — 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 1 is a connected graph, i.e., all nodes in G are connected.

□

**Definition: $\mathbf{TSJ_1}$** — A $TSJ_2$ query is also a $TSJ_1$ query if the join graph G constructed using

[1] 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.

Algorithm 1 is a *fully connected* graph. In other words, for all i and j such that $1 \leq i \leq m$, $1 \leq j \leq m$ and $i \neq j$,

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

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 join condition $P(R_1, \cdots, R_m)$, all participating tuples ($r_k$'s) must span a common time point. □

$TSJ_1$ and $TSJ_2$ are classes of temporal pattern queries (e.g., multi-way temporal joins) in which the lifespans of tuples intersect. For example, Cartesian products across multiple relations (i.e., no join predicates) and a query with the join condition "$R_i.TE<R_j.TS$" are examples of queries that do not belong to either $TSJ_1$ or $TSJ_2$. The characteristics of these two types of queries are crucial in developing the data stream indexing scheme to be described in a later section. We also find it convenient to define a special subclass of $TSJ_1$ queries as follows.

**Definition: $TSJ'_1$** — A query $Q \equiv \sigma_{P(R_1,\cdots,R_m)}(R_1, \cdots, R_m)$ is a $TSJ'_1$ query if

> Q is a $TSJ_1$ query, and all comparison predicates (not join predicates) in P involve only non-time attributes.
>
> □

The class of $TSJ'_1$ join queries includes the "natural time-join" [Cli85, Cli87], the "intersection join" [Gun91], and the temporal join operators discussed in [All83, Leu90] and in a later section: contain-join(X,Y), overlap-join(X,Y), and intersect-join(X,Y).

## 3 Timestamps vs Ordinary Attributes

From a theoretical view point, there is no fundamental difference between timestamps and integer valued attributes such as salary and department number. However there are significant practical distinctions with respect to the manner in which temporal data is updated and queried. Some of the distinctions that we list have been pointed out by other researchers, and the list is not necessarily complete. We believe however that the list does represent the major distinctions.

1. Time is advancing in one direction.
   The time domain is continuously expanding and the most recent time point is the largest value in the domain.

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 [Leu90].

3. Types of query may be different.

   Temporal queries share many common operators with conventional queries. The following highlights the major differences which are more in the nature of characterizing the types of temporal query 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.

   - The join condition often contains a number of inequality join predicates on only timestamps.
   - A special kind of select query, commonly called *snapshot* query, allows us 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 of the time an event X occurs.

   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, 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 was pointed out in [Cli87, Seg87] that the lifespan may be changed as a result of temporal qualification. In this paper, meta-data is not our major concern; see [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 never 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])[2]. In most work, the so-called "append-only" policy is adopted:

   > The *current* data value (of attribute V) of an object s is represented by a tuple '<s,v,$t_s$,*now*>'. That is, the value v is valid since $t_s$. The tuple is called *current* tuple. When the value v is no longer valid at $t_e$, the TE attribute of the tuple (i.e., "*now*") is updated to $t_e$. The tuple '<s,v,$t_s$,$t_e$>' is called a *history* tuple.

   That is, users cannot update the timestamps arbitrarily but users can query timestamps. Coupled with the fact that time is advancing in one direction, this kind of update suggests

[2] Proactive update is also another proposed feature that is seldom found in conventional database systems.

that a special storage structure that exploits the "append-only" policy may be more efficient (e.g., [Ahn88, Gun89])[3].

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[4]. For this reason, the comparison between "*now*" and any other data values have to be defined accordingly. For example, one has to define what the 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. Note that the TS and 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 analysis [Kab90, Nar89][5].

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

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

[4] 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.

[5] 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.

## 4 Temporal Operators

In this section, we discuss several temporal operators that are commonly used in temporal DBMS literature; they are temporal join, select, "within", "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. In other words, most temporal operators are syntactic sugar — they can be directly specified in terms of comparison predicates and join predicates involving only timestamps; the use of these operators allows us 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 [Sel79]. Moreover, the query processing strategies that are implemented are based on what are expected to be the common types of query and data characteristics. It is our belief that a major difference between temporal and conventional queries is in the types of query that are common. Although we can translate temporal queries into their equivalent conventional counterparts, executing the translated queries on conventional DBMS may be very inefficient because the common forms of translated queries are often ignored by the conventional relational query processors and optimizers. Giving attention to the characteristics of temporal queries is therefore key to an efficient query processing algorithm and optimization. This is the focus of the remainder of this paper.

### 4.1 Join Operators

In [Leu90], we noted that temporal join operators often contain inequality predicates involving only timestamps. For example, the following temporal join operators represent the thirteen temporal relationships presented in [All83]:

meet-join(X,Y) — "X.TE=Y.TS"
contain-join(X,Y) — "X.TS<Y.TS ∧ Y.TE<X.TE"
start-join(X,Y) — "X.TS=Y.TS ∧ X.TE<Y.TE"
finish-join(X,Y) — "X.TS>Y.TS ∧ X.TE=Y.TE"
equal-join(X,Y) — "X.TS=Y.TS ∧ X.TE=Y.TE"
overlap-join(X,Y) — "X.TS<Y.TS ∧ Y.TS<X.TE ∧ X.TE<Y.TE".

Note that the overlap-join(X,Y) is asymmetric with respect to the operands. 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 (denoted as TE-join), which is also called the natural time-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 on only non-time attributes hold."

The TE-join is a T-join when the equality join predicate P is "true". In [Gun91], the authors 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:

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

where X(S,V,TS,TE) and Y(S,U,TS,TE) are temporal relations, L is the projection list involving only non-time attributes (i.e., X.S, X.V, and Y.U), and P is the join predicate involving only non-time attributes (i.e., "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):

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

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:

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

Note that the TE-outerjoin is not the same as the "outerjoin" operator defined in [Cod79]. As in the case for the TE-join, the TE-outerjoin can also be defined in terms of traditional relational algebraic operators. The equivalent form in the relational tuple calculus is presented in Appendix A.

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 the following subsection which concerns snapshot operators.

## 4.2 Snapshot Operators

We discuss several commonly used snapshot operators — **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:

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

  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:

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

  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 we 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:

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

In short, the snapshot operators are equivalent to a conjunction of several comparison predicates involving only timestamps. As an example, the following query selects tuples that satisfy a predicate P on a non-time attribute during the entire interval $[t_1, t_2)$:

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

## 4.3 "Within" Operator

There are two "within" operators which are distinguished by the combination of operands: time intervals and time points. We define the **within-i-i** operator which represents the maximum "distance" (N) between two time intervals $[t_1, t_2)$ and $[t_3, t_4)$ as follows:

**within-i-i**($[t_1, t_2)$,$[t_3, t_4)$,N) holds if

- "$[t_1, t_2)$ **intersect** $[t_3, t_4)$" holds, *or*
- "$(0 \leq t_3 - t_2 < N) \vee (0 \leq t_1 - t_4 < N)$" holds.

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.

The **within-i-p** operator which represents the maximum "distance" between a time interval [ts,te) and a time point t is defined as follows:

**within-i-p**([ts,te),t,N) holds if

- "t **between** [ts,te)" holds, *or*
- "$0 \leq t-te < N$" or "$0 \leq ts-t \leq N$" holds.

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 a time-point temporal relation Y(S,V,T), the **within-i-p**(X,Y,N) operator becomes a join operation whose join condition is "$X'.TS \leq Y.T \wedge Y.T < X'.TE$".

## 4.4 Time-project & Time-union Operators

The **time-project** operator (denoted as $\pi_T$) 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]. As an example, the following query retrieves the time interval(s) during which Tom was the manager of the Sales Department from the relation DEPT(Dname,Mgr,TS,TE):

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

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,

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

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 unions several overlapping intervals and returns a single equivalent interval. Second, the time-union operator can play a role in query optimization if the result from the time-project operator is joined with other temporal relations. However, the time-union operator is really a fixed point computation and cannot be expressed in terms of traditional relational algebra[6]. 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[7]:

```
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, Te<TE.
overlap(Ts,Te) :- r(TS,TE), TS<Ts, Ts≤TE, TE<Te.
```

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

[7] 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 — see Section 7.

In a later section, we discuss the optimization issues raised by the use of various temporal operators. First, we present a *stream processing* approach for 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.

## 5 Stream Processing Techniques

We present stream processing algorithms for implementing temporal join and semijoin operations. For properly sorted streams of tuples, we show that temporal join and semijoin operators can often be carried out with a single pass over the input streams, and the amount of workspace required can be small. The tradeoffs between sort orders, the amount of local workspace, and multiple passes over input streams are discussed.

### 5.1 What is Stream Processing?

Abstractly, a *stream* can be defined as an ordered sequence of data objects. Stream processing is a paradigm which has been widely studied [Abe85, Par90] and used in languages such as Lisp; it is very similar to list processing in which elements of a list are sequentially processed. Stream processing also appears naturally in database systems; it closely resembles the notion of dataflow processing. In the functional data models [Shi81, Bat88] a function, which is implemented by a stream processor, is a mapping from input stream(s) into output stream(s). Furthermore, function composition can be viewed as "connecting" a network of stream processors through which data objects flow.

A classical example of a stream processing operation is the merge-join. When we merge-join two relations sorted on their key attribute, at any point in time we need only one tuple from each table as the "state". The join is efficiently implemented as both tables are read only once. Moreover, the output from this join operation is also sorted on the key attribute so that subsequent operations on this output can then take advantage of this ordering [Smi75, Sel79].

There are several intrinsic characteristics of stream processing in database systems. First, a computation on a stream can access only one element at a time (referenced via a *data stream pointer*) and only in the specified ordering of the stream. Second, the implementation of a function as a stream processor may require keeping some local state information in order to avoid multiple readings of the same stream. The state 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, average, 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. That is, a stream processor takes an object from each input stream and, depending on the current state, it can change the current state to a new state and at the same time output some objects on its output stream(s).

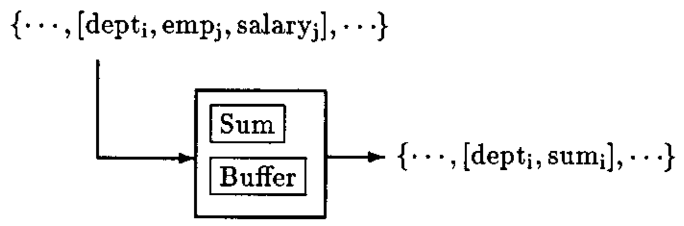
$\{\cdots, [\mathrm{dept_i}, \mathrm{emp_j}, \mathrm{salary_j}], \cdots\}$
Sum
Buffer
$\{\cdots, [\mathrm{dept_i}, \mathrm{sum_i}], \cdots\}$

Figure 2: A stream processor to sum all employees' salaries in each department.

Let us consider a simple stream processor which lists all the departments and computes the sum of all employees' salaries in each department, as shown in Figure 2. If the stream of tuples is grouped by the department name, the local workspace simply contains the partial sum and a buffer for the tuple just read. The point here is that the state contains summary information, and the function (i.e., sum) is expressed in terms of the current state and an input object.

The third characteristic of stream processing is that there are often tradeoffs among the following factors:

1. the size of the local workspace which depends on the function itself, the statistics of a specific instance of the data streams, and the garbage-collection criteria,
2. the sort ordering of input streams, and
3. multiple passes over input streams (i.e., the number of disk accesses).

Very often stream processing requires input streams to be properly sorted in order to perform the computation while reading the input streams only once. In addition, the sort ordering of input streams greatly affects the size of the 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.

For many practical situations in query processing, it is important to make use of the ordering of tuples so that we can minimize the amount of the local workspace and the number of passes over the input streams. As temporal data often implies ordering by time, treating temporal relations as ordered sequences of tuples (i.e., streams of tuples) suggests that stream oriented strategies for temporal query processing could be especially effective. In the next subsection we discuss the application of stream processing algorithms to implementing temporal join and semijoin operations. In these discussions the sort ordering of streams plays a major role.

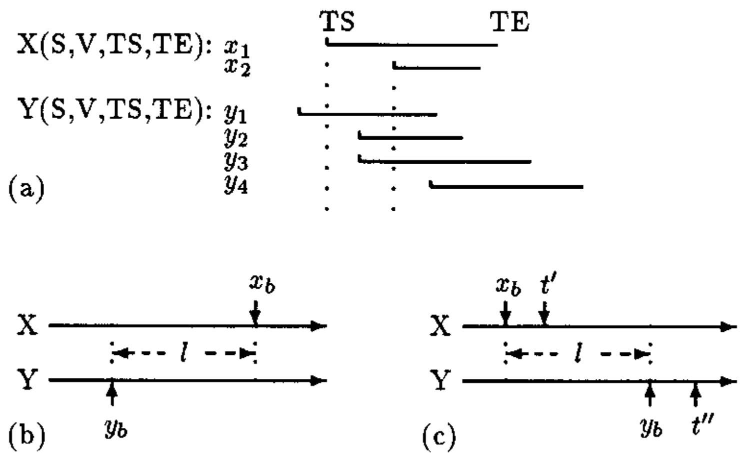

Figure 3: Contain-join: Both X and Y are sorted on TS in ascending order (only timestamps are shown)

### 5.2 Sort Orderings

Suppose we have temporal relations X(S,V,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 in the stream processing paradigm. We concentrate only on inequality joins, such as the contain-join, that is, the operators that have only inequalities in their explicit constraints. We focus on how various sort orderings would affect the size of the local workspace required for the operations. Before we proceed, we note that the only form of state information we need consider for joins and semijoins is the subsets of the tuples previously read and not any aggregate information such as sum, max, avg, etc.

#### 5.2.1 Contain-join

*Contain-join*(X,Y) outputs the concatenation of tuples X and Y if the lifespan of X contains that of Y; that is, the join condition is "X.TS<Y.TS $\wedge$ Y.TE<X.TE".

The join algorithm assumes that: (1) there is an input buffer for reading tuples from each stream (denoted as <Buffer-x, Buffer-y>, and the tuples as $x_b$ and $y_b$), and (2) on the average, the TS (and TE) values of two consecutive X tuples differ by $1/\lambda_x$ units of time (similarly, $1/\lambda_y$ for Y tuples). The algorithm for the case when both relations X and Y are sorted on the attribute TS in ascending order, as shown in Figure 3(a), is:

1. Initially there is no state tuple and the first tuple from each stream is read and stored in the buffer.

2. Read phase: copy $x_b$ and $y_b$ into the state space. Reading next tuples from both streams depends on the TS values of $x_b$ and $y_b$. The first case is when "$y_b$.TS $<$ $x_b$.TS" as shown in Figure 3(b). As all Y tuples read so far do not join with $x_b$, more Y tuples should be read such that "$y_b$.TS$\geq$$x_b$.TS".

   The second case is when "$y_b$.TS$\geq$$x_b$.TS" as shown in Figure 3(c). The state of the current computation is:
   {X tuples whose lifespan span $y_b$.TS}
   $\cup$ {Y tuples whose TS value lies in $l$}.
   A tuple from an input stream which allows more state tuples to be discarded will be read. To estimate the number of disposable state tuples, $1/\lambda_x$ and $1/\lambda_y$ are used. If the next X tuple is read, disposable Y tuples are those which satisfy "$x_b$.TS $\leq$Y.TS$\leq\overline{t'}$ ", where the expected value of $t'$ (denoted as $\overline{t'}$) is ($x_b$.TS + $1/\lambda_x$). Likewise, disposable X tuples are those which satisfy "$y_b$.TS $\leq$ X.TS$\leq\overline{t''}$ " when the next Y tuple is read, where $\overline{t''}$ is ($y_b$.TS+$1/\lambda_y$).

3. Garbage-collection phase: discard X tuples in the state if "X.TE<$y_b$.TS". Also discard Y tuples if "Y.TS<$x_b$.TS". The garbage-collection conditions must guarantee that the Y and X tuples being discarded do not satisfy the join condition with any subsequent X and Y tuples respectively.

4. Join phase: output X and Y tuples if they satisfy the join condition.

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

Note that 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 merged together to gain better performance. Also, the state can be characterized as follows: (1) when there is no Y tuple in the state, the maximal set of X tuples that are required to be kept in the state consists of all overlapping X tuples at time point $y_b$.TS, and (2) conversely, when 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 the attribute 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: the Y tuples that can be discarded when an X tuple is read would be the same as above, but the disposable X tuples when the next Y tuple is read are those which satisfy "$y_b$.TE$\leq$X.TE$\leq$$y_b$.TE+$1/\lambda_y$".

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

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

### 5.2.2 Contained-semijoin & Contain-semijoin[8]

*Contained-semijoin*(X,Y) selects X tuples if there *exists* a Y tuple such that the lifespan of Y contains that of X. *Contain-semijoin*(X,Y) selects those X tuples whose lifespan contains that of any Y tuple. For semijoins, a stream processor can output a tuple as soon as it finds the first matching tuple. Because of this, we devise an optimized algorithm which requires *only one buffer* for each input stream. Suppose the relation X is sorted on attribute TS and the relation Y is sorted on TE in ascending order. The algorithm for contain-semijoin(X,Y) (and contained-semijoin(Y,X) respectively) is as follows:

1. Read an X tuple and store it as $x_b$.
2. Read the next Y tuple and store it as $y_b$ (the previous $y_b$ is discarded) until one of the following holds:
   - "$x_b$.TS<$y_b$.TS $\wedge$ $y_b$.TE<$x_b$.TE" — i.e., $x_b$ and $y_b$ satisfy the semijoin condition, or
   - "$y_b$.TE$\geq x_b$.TE", or
   - all Y tuples have been read.

   If "$y_b$.TS$\leq x_b$.TS", immediately go to Step 2. On the other hand, if the semijoin condition is satisfied between $x_b$ and $y_b$, output $x_b$. (For contained-semijoin(Y,X), $y_b$ is output if the condition is met and go to Step 2). It can be easily verified that only one Y tuple needs to be kept in the workspace.
3. Go to Step 1 unless the termination condition is met.

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 a number of "dangling" tuples may be eliminated, which may reduce the size of workspace for join operations. In Table 1 we summarize the effect of various sort orders on the contain-join(X,Y), contain-semijoin(X,Y) and contained-semijoin(X,Y). The readers may refer to [Leu90] for the state information requirements of processing other inequality joins and semijoins for other sort ordering combinations.

## 6 Generalized Data Stream Indexing

We now turn to the processing of the *snapshot queries* using a new indexing technique based on $\mathrm{TSJ}'_1$ queries, i.e., multi-way temporal joins. We focus on two way joins, i.e., two data streams, but the results are easily generalized to handle more than two data streams. We then discuss the query

[8] 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 | $\uparrow$ | TS | $\uparrow$ | (a) | (c) | (c) |
| TS | $\downarrow$ | TS | $\downarrow$ | - | - | - |
| TS | $\uparrow$ | TE | $\uparrow$ | (b) | (d) | - |
| TS | $\downarrow$ | TE | $\downarrow$ | - | - | (d) |
| TE | $\uparrow$ | TS | $\uparrow$ | - | - | (d) |
| TE | $\downarrow$ | TS | $\downarrow$ | (b) | (d) | - |
| TE | $\uparrow$ | TE | $\uparrow$ | - | - | - |
| TE | $\downarrow$ | TE | $\downarrow$ | (a) | (c) | (c) |

| | |
|---|---|
| $\uparrow$ | Sorting the corresponding attribute in ascending order. |
| $\downarrow$ | Sorting the corresponding attribute 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} $\cup$ {Y tuples whose TS value lie in $l$} |
| (b) | state = {X tuples whose lifespan span $y_b$.TE} $\cup$ {Y tuples whose lifespans are contained within $l$} |
| (c) | state $\subseteq$ {X tuples whose lifespan span $x_b$.TS} $\cup$ {Y tuples whose TS values lie in $l$} |
| (d) | local workspace = <Buffer-x, Buffer-y>. |

Table 1: Effect of various sort orders on contain-join, contain-semijoin & contained-semijoin

processing algorithms using data stream indices. A quantitative analysis and a comparison with conventional indexing schemes are presented.

## 6.1 Data Streams

As we discussed earlier, a stream is an ordered sequence of data objects and temporal data often implies ordering by time. In the "append-only" databases, two natural situations occur in which history tuples can be organized as data streams:

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 timestamp of the tuple in the data stream is then modified to $t_e$. In this approach tuples in the data stream 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 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 timestamp of the tuple is modified to $t_e$: the history tuple 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.

Data streams 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 using the data stream pointer. To simplify our discussion, we focus only on data streams that are sorted on the TS timestamp. The schemes can be easily adapted to the case when the data streams are sorted on the TE timestamp; see [Leu92] for details.

## 6.2 Checkpointing Query Execution

Consider the execution of query Q $\in$ $TSJ'_1$ as a stream processor. In our approach indices are built by periodically checkpoint the execution of Q on X and Y along the time axis, and checkpoints are in turn indexed on their checkpoint times as depicted in Figure 4. Informally, a checkpoint (e.g., $ck_{t_2}$ in Figure 4) at a time point (e.g., $t_2$) contains enough information about the execution of Q on X and Y such that the response of a snapshot 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 $ck_{t_2}$) using the time index on checkpoints, then access tuples in the operand data streams which started since $t_2$. "Continue" the execution of the query (e.g., in this case Q) using the tuples thus accessed until t.

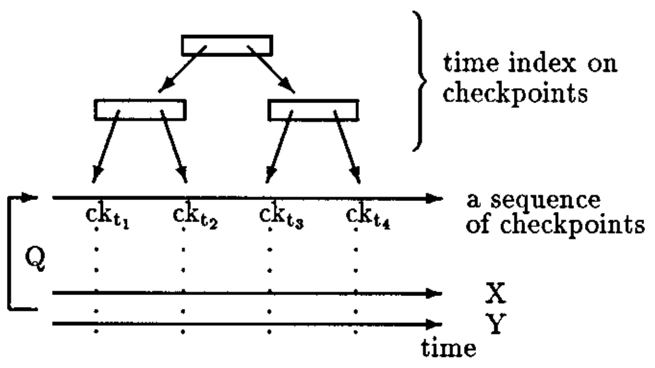

Figure 4: Checkpointing a query execution and time index on checkpoints

Since not all tuples of the operand data streams can be accessed randomly, 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. For convenience, we refer to the query Q as the indexing condition[9].

We now discuss how the checkpointing is performed. Suppose we have an indexing condition $Q \equiv \sigma_P(X,Y) \in TSJ'_1$. We first derive a new predicate Px (called the *state predicate*) from P by replacing all terms in P involving Y with "true". That is, Px contains only comparison predicates involving X. The state predicate Px becomes the indexing condition on the data stream X. Similarly, we derive the state predicate Py for the data stream Y. As will become clear shortly, the checkpoint of Q at a time point t can be expressed in terms of the checkpoint of the derived state predicates on individual data streams at time point t.

Three kinds of information are stored in a checkpoint (denoted as ck) — *checkpoint time*, *state information* and *data stream pointers*. For a checkpoint $ck_t$ performed at time point t, let the checkpoint prior to $ck_t$ be denoted as $ck_{t^-}$ (at time $t^-$) and define[10]:

1. The checkpoint time is t[11].
2. The state information of the data stream X, denoted as $s_t(X)$, contains the tuple identifiers (TID's) of all tuples $x \in X$ such that "x.TS<t $\wedge$ t$\leq$ x.TE $\wedge$ Px" holds, where Px is the state predicate. Basically the state information contains tuples which are active as of the checkpoint time and satisfy the state predicate. Note that tuples in $s_t(X)$ either belong to $s_{t^-}(X)$ or start during the interval $[t^-,t)$. Similarly, the state information of a data stream Y contains tuples

[9] More generally, the indexing condition Q can be a query that subsumes a set of frequently asked queries. For example, Q can be $\sigma_{\text{intersect-join}(X,Y)}(X,Y)$.
[10] It follows that $t^- < t$. If there is no such $ck_{t^-}$, $ck_{t^-}$ and $t^-$ are assumed to be an empty set and 0 respectively.
[11] The special marker (*now*) is the latest current time and thus it represents a time point larger than all checkpoint times.

$y \in Y$ such that "$y.TS<t \wedge t \leq y.TE \wedge Py$" holds, where Py is the state predicate for the data stream Y.

3. The data stream pointer of X, denoted as $dsp_t(X)$, 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. Using the data stream pointer, one can access the tuples which start at t or after t. Similarly, the data stream pointer of Y at checkpoint time t is denoted as $dsp_t(Y)$.

In addition to the three basic types of checkpoint information mentioned above, one can also store the TID's of matching tuple pairs as the *incremental result* in checkpoints. Given a checkpoint $ck_t$ (at time t) and its next checkpoint $ck_{t^+}$ (at time $t^+$), we denote $X_t$ and $Y_t$ as the portion of data streams X and Y respectively that start during $[t,t^+)$. Note that $X_t$ and $Y_t$ can be accessed via the data stream pointers $dsp_t(X)$ and $dsp_t(Y)$ respectively. We also denote $S_t(X)$ and $S_t(Y)$ as the tuples retrieved using the TID's in the state information $s_t(X)$ and $s_t(Y)$ respectively.

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

$$\sigma_P(\ (X_t \cup S_t(X)),\ (Y_t \cup S_t(Y))\ )$$

that is, the tuple pairs that satisfy the join condition. Note that a pair of TID's need not be stored if both TID's have been stored in the state information at checkpoints $ck_t$ or $ck_{t^+}$. When we discuss the query processing algorithms in the next section, we will focus on the three basic types of checkpoint information which can be used to compute the incremental results.

Example 1: Consider $Q_2 \equiv \sigma_{\text{intersect-join}(X,Y)}(X,Y)$ as the indexing condition (Figure 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. 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 Px and Py are "true" for X and Y respectively.

In Table 2 we list the three types of information in the checkpoints as well as the incremental result that can be stored in checkpoints. □

At this point, we can make some comments regarding the basic scheme. First, given a sequence of checkpoints as illustrated in Figure 4, 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 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.

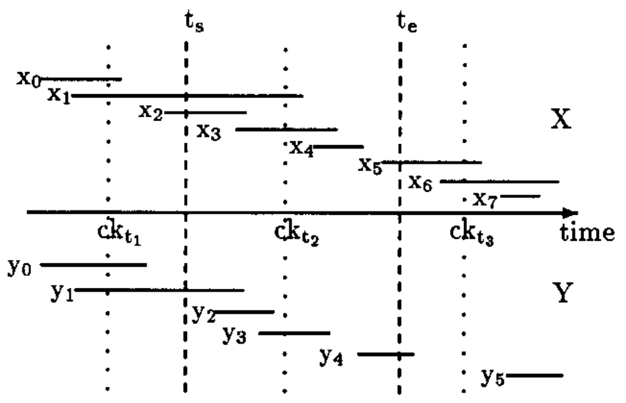

Figure 5: Checkpointing the query $Q_2$

| checkpoints | $ck_{t_1}$ | $ck_{t_2}$ | $ck_{t_3}$ |
|---|---|---|---|
| t | $t_1$ | $t_2$ | $t_3$ |
| $s_t(X)$ | $\{x_0,\ x_1\}$ | $\{x_1,\ x_3\}$ | $\{x_5,\ x_6\}$ |
| $s_t(Y)$ | $\{y_0,\ y_1\}$ | $\{y_3\}$ | $\{\ \}$ |
| $dsp_t(X)$ | $\{x_2\}$ | $\{x_4\}$ | $\{x_7\}$ |
| $dsp_t(Y)$ | $\{y_2\}$ | $\{y_4\}$ | $\{y_5\}$ |
| $ir_t(Q_2)$ | $\{ <x_1,y_2>, <x_2,y_1>, <x_2,y_2>, <x_3,y_1>, <x_3,y_2> \}$ | $\{ <x_4,y_3>, <x_4,y_4>, <x_5,y_4> \}$ | $\{\ \}$ |

Table 2: Checkpoints of $Q_2$ in Figure 5

## 6.3 Query Processing using Data Stream Index

In this subsection, we discuss the processing algorithms for some types of complex snapshot 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[12]:

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

In order to use the data stream index, 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"[13]. Furthermore, we require that $P'|_x \Rightarrow Px$ which is the state predicate that is used to determine and store the state information of data stream X. Similarly, we require that $P'|_y \Rightarrow Py$. 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 the query processing. The algorithm that uses the data stream index for the **intersect** queries is as follows.

Algorithm using Data Stream Index

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.
2. Retrieve the tuples using TID's in the state information $s_t(X)$ and $s_t(Y)$ that are stored in the checkpoint $ck_s$, and apply the predicates $P'|_x$ and $P'|_y$ respectively.
3. Retrieve tuples in X and Y which start during $[t,t_e)$ by following the data stream pointers $dsp_t(X)$ and $dsp_t(Y)$, and apply the predicates $P'|_x$ and $P'|_y$ respectively.
4. The set of all tuples from (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 held in workspace is limited to tuples spanning a common point in time.

For the **as of** queries, the query processing algorithm remains essentially the same except that in step 3 only tuples in X and Y which start during $[t,t_s]$ (instead of $[t,t_e)$) are accessed.

It can be shown that the following classes of queries can also be processed using the data stream indices:

[12] We believe that the class of queries that can be processed using data stream indices can further be generalized — the investigation will be left as a future research work.

[13] More restrictive predicates ($P'|_x$ and $P'|_y$) may be obtained by using constraint propagation algorithms [Ull82, Chak84, Jar84, Leu91].

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

Example 2) Consider the data stream index (whose checkpoints are shown in Figure 5 and Table 2) and a user query $\sigma_{\text{intersect-join(X,Y)}}(X,Y)$ **intersect** $[t_s, t_e)$. In step (2), we retrieve tuples $\{x_0, x_1\}$ and $\{y_0, y_1\}$. By following the data stream pointers, 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 now discuss some limitations of the proposed checkpointing and indexing scheme. In the proposed scheme, only $TSJ'_1$ queries can be allowed as the indexing conditions. Recall that for join queries in $TSJ'_1$, the lifespans of all participating tuples have to intersect with each other. To understand the importance of this restriction, let us consider the "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[14]. For the query processing algorithms that we 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 to $TSJ'_1$ queries, we only need to store in a checkpoint the TID's of tuples that span the checkpoint time.

The state information may still require a large amount of storage space when many qualified tuples span the checkpoint times. In [Leu92a] we presented some optimization techniques in reducing the storage requirement and discussed the tradeoffs. In the following subsection, we present a quantitative analysis on the overhead of storing the state information in checkpoints of the original scheme.

## 6.4 Quantitative Analysis

We first list some required notation:

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

[14] 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.

- $\mathrm{size_{tid}}$ 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 is:

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

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 $\sigma_\mathrm{q}$, i.e., $\sigma_\mathrm{q}$ 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., $\sigma_\mathrm{q}$ is smaller).

## 6.5 Comparisons

Several temporal indices have recently proposed (e.g., [Elm90, Gun89, Kol89, Kol91, Lom89, Rot87]) which are extensions of traditional dense indexing methods such as $\mathrm{B^+}$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 data stream indexing technique. For example, suppose we create a $\mathrm{B^+}$tree index on the TS timestamp. That is, there is an index entry in the $\mathrm{B^+}$tree for every tuple in the relation. Recall that the relation lifespan is $\mathrm{TR_{ls}}$ and the rate of insertion of tuples is $\lambda$. The total number of TID's stored in the leaf nodes of the $\mathrm{B^+}$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 ($\sigma_q$) is 1. The number of TID's stored in the state information of all checkpoints is:

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

where $n_{ck}$ is the number of checkpoints that have been taken [15]. If we perform the checkpointing at a rate smaller than $1/\overline{T_{ls}}$, the data stream index would require less storage space. More detailed analysis can be found in [Leu92a].

## 6.6 Temporal Aggregate Functions

The notion of state information is not limited to the qualified tuples that span checkpoint times, 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. It may generally 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 on the daily sales records for the year of 1989 [16]. We further suppose that we checkpoint the aggregate query on a yearly basis. For example, consider the checkpoint times $t_{i-1}$ and $t_i$. The state information at checkpoint time $t_i$ can be defined as TID's of tuples that started during the last week of the year of $t_{i-1}$. To process the above aggregate query, one can retrieve the state information at $t_i$ (i.e., for the year of 1989) and tuples that started after $t_i$, and evaluate the aggregate function. Note that the number of TID's in the state information for this aggregate query is bounded — at most one week of daily sales records. Also note that the temporal data is not necessarily time-interval tuples.

# 7 The "interval" Operator

In this section, we briefly discuss the interval operator, its role in processing temporal join operations, and its implementation as a stream processor.

We first define a generalized time-interval temporal relation as $E(S,V_1,\cdots,V_k,TS,TE)$ for some $k \geq 1$, where S is the surrogate, $V_i$'s ($1 \leq i \leq k$) are time-varying attributes, and the interval [TS,TE) denotes the lifespan of a tuple [17]. For example, the relation E can be the result of the intersection-

[15] 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.

[16] 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."

[17] That is, each tuple has only two timestamps which indicate the lifespan.

join(X,Y) defined in [Gun91]. We can now define the **interval** operator in terms of the **time-union** and project operators as follows:

$$\textbf{interval}(E) \equiv \textbf{time-union}(\ \pi_{TS,TE}(E)\ )$$

That is, the **interval** operator unions the lifespans of all tuples in E and returns one or more non-overlapping time intervals.

To illustrate the usefulness of the **interval** operator, let us consider the following $TSJ'_1$ join query involving relations X(S,V,TS,TE), Y(S,U,TS,TE) and Z(S,W,TS,TE):

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

In general, there are a number of equivalent query plans from which the best plan will be chosen for execution. One possible query plan (which is illustrated in Figure 6) is to use the **interval** operator which computes the non-overlapping time intervals during which there exist X tuples that satisfy the predicate "$X.V=v$". These time intervals can then be used to facilitate the join between relations Y and Z. The query plan can be summarized as follows. First, the tuples that satisfy "$X.V=v$" are retrieved via scanning the relation or accessing a conventional index on the attribute V, if it is available. The **interval** operator is then applied to the qualified tuples to obtain their unioned "lifespans". Let us denote the time-unioned intervals as $\{\ [t_1, t_1^+), \cdots, [t_k, t_k^+)\ \}$, for some $k \geq 1$. Note that the tuples in relations Y and Z that satisfy the join condition must also intersect with these time intervals. If a stream index is supported on Y and Z, the join between Y and Z can be processed as follows:

$$\sigma_{\text{intersect-join}(Y,Z)}(Y,Z)\ \textbf{intersect}\ [t_i, t_i^+) \qquad\qquad \text{for } 1 \leq i \leq k.$$

The final query response is the join between the temporary results from the above join operations and the tuples in relation X that satisfy "$X.V=v$".

The central part of the implementation of the **interval** operator is the **time-union** operator. In Figure 7, we show a stream processing implementation of the **time-union** operator. In this implementation, the input time intervals are sorted on the TS timestamp in ascending order. The stream processor keeps the most recently read tuple (denoted as $x_b$) in a buffer space. The state information at any point in time is the minimum value of the TS timestamp (denoted as $TS_{min}$) and the maximum value of the TE timestamp (denoted as $TE_{max}$) of the tuples that overlap with each other and have been read thus far (their initial values are 0). If the tuple $x_b$ has the TS value greater than $TE_{max}$, the stream processor outputs a pair of values — $[TS_{min}, TE_{max})$ and keeps the $x_b$.TS and $x_b$.TE values as $TS_{min}$ and $TE_{max}$ respectively. Otherwise, the stream processor will keep the larger of the two values: $x_b$.TE or $TE_{max}$, as the new $TE_{max}$. As in other stream processors that we discussed earlier, the sort ordering of input data plays an important role in its efficient implementation.

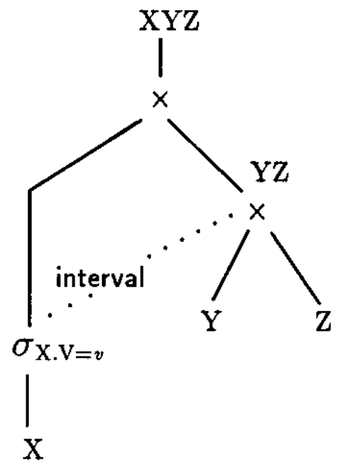

Figure 6: A query plan using the **interval** operator

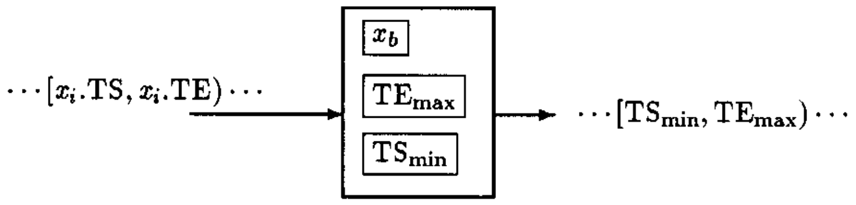
$\cdots[x_i.\mathrm{TS}, x_i.\mathrm{TE})\cdots$
$x_b$
$\mathrm{TE}_{\max}$
$\mathrm{TS}_{\min}$
$\cdots[\mathrm{TS}_{\min}, \mathrm{TE}_{\max})\cdots$

Figure 7: The **time-union** stream processor

## 8 Conclusions & a Look into the Future

In this paper, we discussed several characteristics of temporal data and queries, and presented stream processing techniques for processing temporal join and semijoin operators which often contain a conjunction of a number of inequalities. The effect of sort orderings of streams of tuples on the efficiency with which an operator is implemented and the local workspace requirement in the stream processing environment were studied. An interesting observation is that the optimal sort order may depend on the query itself and the statistics of data instances. Based on the stream processing paradigm, we further proposed a generalized data stream indexing technique that can facilitate the processing of complex snapshot queries.

There are many research issues that need to be investigated. One of the most important one 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 buffer space, 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. Following is an example.

To illustrate the issues involved, which are peculiar to temporal data and queries, let us consider a query $\sigma_P(X,Y,Z)$ where P is "contain-join(X,Y) $\wedge$ contain-join(Z,Y) $\wedge$ $P_y(Y)$" and $P_y(Y)$ is a comparison predicate on relation Y. There are generally many equivalent query plans; we show a typical one in Figure 8(a) which joins relations X and Y first, followed by the join between the intermediate result and the relation Z. In addition to determining the optimal join ordering, the estimation of the size of the intermediate join result, and the implementation of the select predicate $P_y(Y)$ (e.g., by indexing or file scanning), there are several choices which are more peculiar to temporal databases

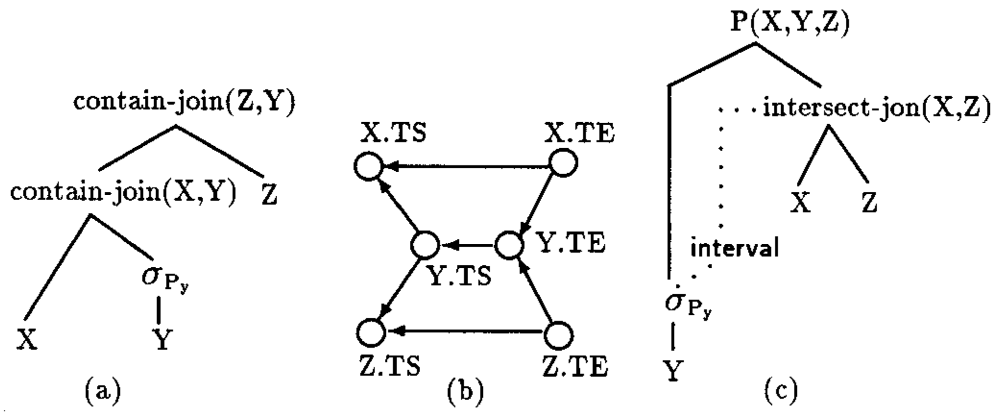

Figure 8: Global optimization: P(X,Y,Z) ≡ contain-join(X,Y) ∧ contain-join(Z,Y)

that the query optimizer has to consider. For example, both joins are contain-joins which can be implemented either as stream processors, which we discussed earlier, or using the nested-loop join method. Using the stream processing approach, one alternative is to sort the relations X and Z on the TS timestamp in ascending order while sorting the relation Y on either TS or TE timestamp (see Table 1). In addition, the query optimizer can choose a stream processing implementation of the contain-join(X,Y) such that its output is also sorted on the TS or TE timestamp of relation Y. That is, one can directly "pipe" the output to the stream processor that implements the second join, i.e., contain-join(Z,Y), without sorting the intermediate join result. The tradeoff is that the stream processor for the contain-join(X,Y) may require a larger buffer space.

An alternative query plan is as follows. We note that the query qualification is equivalent to:

$$X.TS<Y.TS \wedge Y.TE<X.TE \wedge Z.TS<Y.TS \wedge Y.TE<Z.TE \wedge P_y(Y).$$

Together with the implicit intra-tuple integrity constraints: "X.TS<X.TE", "Y.TS<Y.TE", and "Z.TS<Z.TE", the relationship among the timestamps can be represented by a graph as shown in Figure 8(b) where an arrow represents the relationship "<". From the graph, one can determine that there is an implicit join between relations X and Z: intersect-join(X,Z). That is, for each triplet $<x,y,z>$ where $x \in$ X, $y \in$ Y, and $z \in$ Z that satisfies the query qualification, the lifespans of tuples $x$ and $z$ must intersect. Therefore, the query optimizer can choose to perform the select operation on the relation Y first, as shown in Figure 8(c). The intermediate result from the select operation (and the **interval** operator) can be used to "restrict" the join operation between relations X and Z, for example, using the data stream index on X and Z if it is provided. The final query response is the join between these two intermediate results.

In the above problem formulation, we have ignored the project operator which also can be crucial to generating and choosing an optimal query plan. Other research issues that are of interest include

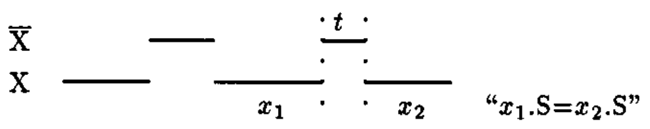

Figure 9: Deriving the relation $\overline{X}$ from X

the following. First, the class of queries that can be processed using the data stream indices is larger than we have presented. Second, with the use of multiprocessor database machines becoming more popular, fragmentation strategies for temporal data as well as temporal query processing and optimization in such systems become more critical to the efficiency and performance. A preliminary study on temporal data fragmentation strategies and temporal query processing algorithms for parallel machines can be found in [Leu91]. Thirdly, in [Leu90] we noted 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.

## Appendix A: The TE-outerjoin

We show that the TE-outerjoin can be defined in traditional tuple calculus. From the relation X(S,V,TS,TE), we first obtain a new relation (denoted as $\overline{X}$) which contains tuples of null values (of attribute V) for each surrogate in X that is not explicitly stored in X, as illustrated in Figure 9:

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

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

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

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

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

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

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

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., $\text{TE}_o$-join) accounts for the case in which a surrogate appears only in the relation X (but not in the relation Y). The joins $\text{TE}_f$-join(X,Y), $\text{TE}_l$-join(X,Y), and $\text{TE}_o$-join(X,Y) are defined as follows:

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

## 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.

[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.

[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.

[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.

[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.

[Gad88] S. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM Trans. on Database Systems*, 13(4):418–448, December 1988.

[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.

[Jar84] M. Jarke. External Semantic Query Simplification: A Graph-theoretic Approach and its Implementation in Prolog. In *Expert Database Systems*, pages 467–482, 1984.

[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.

[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.

[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.

[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. Temporal Query Processing and Optimization in Multiprocessor Database Machines. Technical Report CSD-910077, University of California, Los Angeles, Dept. of Computer Science, November 1991.

[Leu92] T.Y. Leung. *Optimization in Temporal Active Database Systems.* PhD thesis, University of California at Los Angeles, 1992. In Preparation.

[Leu92a] T.Y. Leung and R.R. Muntz. Generalized Data Stream Indexing and Temporal Query Processing. In *2nd Int. Workshop on Research Issues on Data Engineering — Transaction and Query Processing (RIDE-TQP)*, February 1992. Forthcoming.

[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.

[Nar89] S. Narain and J. Rothenberg. A Logic for Simulating Dynamic Systems. In *Proc. of Winter Simulation Conference*, Washington, D.C., 1989.

[Par90] D.S. Parker. Stream Data Analysis in Prolog. In Leon Sterling, editor, *The Practice of Prolog*. The MIT Press, Cambridge, MA, 1990.

[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.

[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.

[Shi81] D.W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Trans. on Database Systems*, 6(1):140–173, March 1981.

[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.

[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.

[Ste86] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, 1986.

[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.

[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.