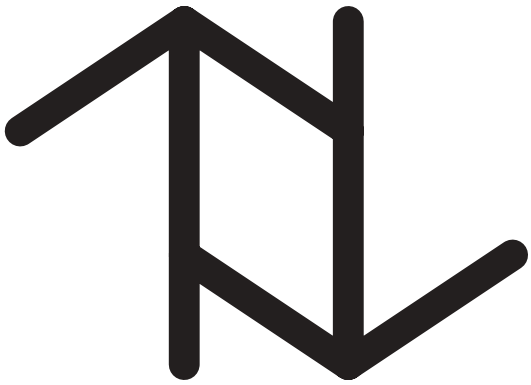


Incremental Join of Time-Oriented Data

Dieter Pfoser and Christian S. Jensen

TR-34

A TIMECENTER Technical Report

Title	Incremental Join of Time-Oriented Data
Author(s)	Dieter Pfoser and Christian S. Jensen
Publication History	September 1998. A TIMECENTER Technical Report.

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector), Michael H. Böhlen, Renato Busatto, Curtis E. Dyreson, Heidi Gregersen, Dieter Pfoser, Simonas Šaltenis, Janne Skyt, Giedrius Slivinskas, Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector), Sudha Ram

Individual participants
Anindya Datta, Georgia Institute of Technology, USA
Kwang W. Nam, Chungbuk National University, Korea
Mario A. Nascimento, State University of Campinas and EMBRAPA, Brazil
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, University of South Florida, USA
Andreas Steiner, TimeConsult, Switzerland
Vassilis Tsotras, Polytechnic University, USA
Jef Wijsen, Vrije Universiteit Brussel, Belgium

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.auc.dk/research/DBS/tdb/TimeCenter/>

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

Data warehouses as well as a wide range of other databases exhibit a strong temporal orientation: it is important to track the temporal variation of data over several months or, often, years. In addition, data warehouses and databases often exhibit append-only characteristics where old data pertaining to the past is retained while new data pertaining to the present is appended. Performing joins on large databases such as these can be very costly, and the efficient processing of joins is essential to obtain good overall query processing performance. This paper presents a sort-merge-based incremental algorithm for time-oriented data. While incremental computation techniques have proven competitive in many settings, they also introduce a space overhead in the form of differential files. However, for the temporal data explored here, this overhead is avoided because the differential files are already part of the database. In addition, data is naturally sorted, leaving only merging. The incremental algorithm works in a partitioned storage environment and does not assume the availability of indices, making it a competitor to sort-based and nested-loop joins. The paper presents analytical cost formulas as well as simulation-based studies that characterize the performance of the join.

1 Introduction

Many databases exhibit an append-only behavior [Cop82]. This occurs when databases capture information about processes. In data warehousing, business processes such as sales or buys are often captured [Kim96]. In scientific applications, physical, chemical, or, e.g., biological processes are monitored [JS94]. Many applications, e.g., financial and medical [PJ98], are faced with accountability requirements that translate into the requirement that all previously current states of the database be retained, which, in turn, dictates an append-only behavior. This paper concerns such databases.

A fundamental and costly operation in any large database, e.g., in a data warehouse, is the join operation [ME92]. Its basic use is to meaningfully combine information distributed over pairs of relations in the database. The cost of a join is directly related to the size of the argument relations, and even with sophisticated join algorithms and indexing techniques, the worst-case cost is $N \times M$, where N and M are the numbers of tuples in the argument relations.

Faced with an expensive operation and potentially very large append-only relations, incremental computation techniques deserve exploration. To compute a join, these techniques assume the availability of the result of a previous computation of the join as well as descriptions of the modifications to the argument relations in-between the time of the previous computation and the current time. If these modifications are relatively small, incremental computation is likely to be very efficient in comparions to recomputation [KD79], [Rou91], [QW91].

This paper presents two join algorithms for append-only relations: a basic sort-merge-based algorithm and its incremental version. The algorithms assume that the relations have associated an interval-valued time attribute. Beyond this distinguished attribute, no assumptions about the numbers of other attributes and their domains are made. For example, additional time-valued attributes may be present. The algorithms exploit the property that the relations in many cases are sorted on their time attribute values. The join predicate is the conjunction of an overlap predicate on the time attribute values and any predicate on the remaining attributes. Hence the only competitor to our algorithms is the nested-loop algorithm.

Incremental computation techniques have proven competitive in many settings. However, they also introduce a space overhead in the form of materialized results and differential files. For the append-only databases explored here, the overhead of differential files is avoided because the differential files are already part of the database. Knowing the time when the previously computed and stored result was computed, the differential files can be extracted from the stored relations. This makes incremental techniques particularly attractive in our temporal setting. The incremental algorithm works in a partitioned storage environment (e.g., [AS88], [Sto87]) and does not assume the availability of indices.

The research on temporal joins can be characterized according to the *reduction criteria* used in the algorithms and by the techniques used for applying the reduction criteria. The reduction criterion in a join is the aspect of the argument relations that is exploited to reduce the number of tuple comparisons and thus to reduce the cost of the join. The criteria are the transaction-time (TT), valid-time (VT), and explicit join attributes (EA) of tuples, and their combinations. Valid time captures when information is true in the modeled reality, and transaction time captures when information is current in the database [SA85]. The join techniques include sort-merge, partition-based, nested-loop, and incremental techniques [ME92]. Of these, the nested loop join applies no reduction criterion. Table 1 gives an overview of research work categorized by the above criteria.

Join technique	**Reduction criterion**		
	TT	VT	VT+EA
Sort-Merge	[LM93] *this work* [GS91]	 [GS91]	[SS93] [GS91]
partition-based		[LM92] [SSJ94]	
Nested Loop			
Incremental	*this work*		

Table 1: Previous Work on Temporal Joins

Leung and Muntz [LM92] describe a partition-based algorithm in a multiprocessor environment. Soo et al. [SSJ94] introduce a partitioning-based algorithm supporting valid time. Segev and Shoshani [SS93] present an algorithm supporting valid time that assumes the argument tuples to be sorted on their join attribute and then on the start of their valid-time attribute, in that order. Both algorithms are suitable for transaction time as well. However, since the properties of transaction time are stricter than the ones that apply to valid time, the algorithms tailored to valid-time data generally leave room for improvement when used for transaction-time data. Leung and Muntz [LM93] present an algorithm that supports transaction time. It is assumed that the relations are sorted on the start of their transaction-time attribute. They do not consider a partitioned environment. Both Segev and Shoshani [SS93], and Leung and Muntz [LM93] also describe their algorithms in an abstract form without implementation details. Performance studies are thus not reported. Gunadhi and Segev [GS91] present sort-based algorithms for joins of temporal relation with and without considering join predicates on the non-temporal attributes. In this work, relations are sorted, depending on the type of join (i.e., only on the temporal attribute, the explicit attribute, or both). Analytical cost formulas and performance studies are reported. However, this work is limited to recomputation, does not contend with partitioned storage, and does not consider the special timestamp *now*.

The outline of the paper is as follows. Section 2 gives the definition of, as well as an algorithm for, a temporal join in a partitioned storage environment. Further, it presents an incremental join algorithm for temporal data. Section 3 shows the cost of the algorithms using analytical formulas, whereas Section 4 presents simulation-based studies that characterize the performance of these join techniques. Finally, Section 5 gives conclusions and directions for future work.

2 Temporal Joins

In this section we define the temporal join, introduce the partitioned storage scheme, and finally present a recomputation, sort-based algorithm and its incremental version for computing the join.

2.1 Temporal Joins and Partitioned Storage

While many temporal aspects of data may be of interest to various applications, we simply make the assumption that relation schemas have at least one time-interval attribute. In addition, we assume that the tuples are inserted in the order of the start times of their intervals. This ordering occurs naturally in many situations, e.g., in data warehousing where business processes are captured over time and in scientific and monitoring applications where chemical, nuclear, or, e.g., biological processes are captured.

We use a common 1NF tuple-timestamped data model for the representation of temporal data. Tuples in a temporal relation have a set of attribute values and a timestamp. We assume the underlying time-line to be partitioned into minimal-duration intervals, termed chronons. The timestamp of a tuple is represented as a single time interval, denoted by inclusive starting and ending chronons. We will assume temporal relational schemas R and S of the format

$$R = (A_1, ..., A_n, \mathrm{T})$$
$$S = (B_1, ..., B_m, \mathrm{T}),$$

where the A_i and B_i are the explicit attributes, and T is the interval-valued timestamp. We will use $\mathrm{T}^{\vdash}$ and $\mathrm{T}^{\dashv}$ to denote start and end times of values of T.

Examples of temporal relation schemas are DeptLocation = (Department, Floor, T) and EmpDepartment = (Employee, Department, T). DeptLocation locates departments at various floors, whereas EmpDepartment assigns employees to departments. Table 1 shows instances of the two schemas. In the examples, we use *now* as a special chronon denoting the present time [CDI+97].

deptLocation

Department	Floor	T
Controlling	1	1 - 2
Controlling	2	2 - 3
Sales	5	3 - 5
Marketing	1	4 - 6
Sales	2	6 - *now*
Controlling	4	7 - *now*
Marketing	5	7 - *now*

empDepartment

Employee	Department	T
Bill	Marketing	2 - 5
Dana	Sales	4 - 6
Siggi	Marketing	5 - *now*
Fox	Sales	6 - *now*
Edgar	Sales	7 - *now*
John	Marketing	9 - *now*

Figure 1: Temporal Relations

Among the temporal aspects that one may associate with data, the aspects of *transaction time* and *valid time* are the most prominent. The valid time of a tuple denotes when the information recorded by the attribute values of the tuple is true in the modeled reality. Transaction time, on the other hand, is system-maintained and captures when tuples are current in the database. Capturing transaction time offers an ideal foundation for supporting accountability requirements to the application at hand.

The transaction time of a tuple is recorded by assigning the time, it is inserted into the database, as the start time of its interval. The end time is the (growing) current time until the tuple is deleted. When that occurs, the time of deletion is assigned to the interval end time. As a result, transaction-time databases satisfy the sequential arrival-order property.

In summary, we assume that the time attribute has the sequentiality property of transaction time, but not necessarily the semantics of it. Rather, it might as well be the case that a natural process, such as incoming bank transactions, creates relations having the properties of transaction time, but the semantics of valid time.

Now, considering the join of temporal relations, let r and s be instances of schemas R and S, respectively. To compute the join of r and s, tuples in r and s have to satisfy the join condition P (snapshot join), and their time intervals have to overlap. The attribute values of the tuple resulting from two qualifying

tuples are the explicit attributes of the two tuples and the overlap of the time intervals. An expression for a temporal join using tuple relational calculus follows.

$$
\begin{aligned}
r \bowtie^T_P s \quad = \quad \{z^{n+m+2} \quad | \quad & \exists x \in r, \exists y \in s \\
& (P(x,y) \wedge \\
& \; z[A_1, ..., A_n] = x[A_1, ..., A_n] \wedge z[B_1, ..., B_m] = y[B_1, ..., B_m] \wedge \\
& \; z[\mathrm{T}] = x[\mathrm{T}] \cap y[\mathrm{T}] \wedge x[\mathrm{T}] \cap y[\mathrm{T}] \neq 0)\}
\end{aligned}
$$

From the expression above, we can see that our join algorithm does not require $\mathrm{T}^{\vdash}$ and $\mathrm{T}^{\dashv}$ to be the *sole* temporal attributes of the relation. Any of the explicit attributes can be temporal, too. Our algorithms will work for any kind of join predicate P. This join is a natural generalization of the conventional join, with join predicate P, on non-temporal relations, in that it is snapshot reducible [Sno87] to this join.

We partition each temporal relation into a current and an old partition. Tuples that fulfill the criterion $\mathrm{T}^{\dashv} = now$, i.e., they are current, are placed in the current partition. Tuples that are logically deleted are placed in the old partition. As mentioned earlier, deleting a tuple means assigning an end-transaction time to the tuple that is different from *now*.

With this partitioning scheme, tuples in the current partition remain ordered by increasing $\mathrm{T}^{\vdash}$, as new tuples with later begin times are appended to the relation. Tuples in the old partition are ordered by increasing $\mathrm{T}^{\dashv}$, as tuples are appended to the old partition when they are logically deleted from the current partition, and the time of deletion is assigned to $\mathrm{T}^{\dashv}$.

Using partitioned relations, a temporal join is computed as the union of the four joins of the current partition relations r_{cur} and s_{cur}, and the old partitionss, r_{old} and s_{old}.

$$r \bowtie^T_P s = r_{cur} \bowtie^T_P s_{cur} \cup r_{cur} \bowtie^T_P s_{old} \cup r_{old} \bowtie^T_P s_{cur} \cup r_{old} \bowtie^T_P s_{old}$$

In Table 2 we split our example relation into a current and an old partition.

Department	Floor	T

$\mathrm{deptLocation}_{old}$

Controlling	1	1 - 2
Controlling	2	2 - 3
Sales	5	3 - 5
Marketing	1	4 - 6

$\mathrm{deptLocation}_{cur}$

Sales	2	6 - now
Marketing	5	7 - now
Controlling	4	7 - now

Employee	Department	T

$\mathrm{empDepartment}_{old}$

Bill	Marketing	2 - 5
Dana	Sales	4 - 6

$\mathrm{empDepartment}_{cur}$

Siggi	Marketing	5 - now
Fox	Sales	6 - now
Edgar	Sales	7 - now
John	Marketing	9 - now

Figure 2: Partitioned Temporal Relations

2.2 Temporal Join Recomputation

In the previous section we defined the temporal join in a partitioned environment. As the next step, we give a recomputation algorithm for computing the join.

The whole recomputation algorithm for the join comprises four sub-join algorithms. The basic idea behind the sub-joins is to exploit the orderings of the tuples with respect to their time attribute values. This is similar to sort-merge joins when relations are presorted. However, the sub-joins differ from sort-merge algorithms in that the predicate on the time attribute values is interval intersection instead of equality.

Figure 3 visualizes the four sub-joins of the temporal join. The tuples of the partitions are represented by their time intervals. Time proceeds from left to right, and the dotted line symbolizes the current time. Tuples are appended at the bottom. In the joins, the tuples are always read from the bottom to the top.

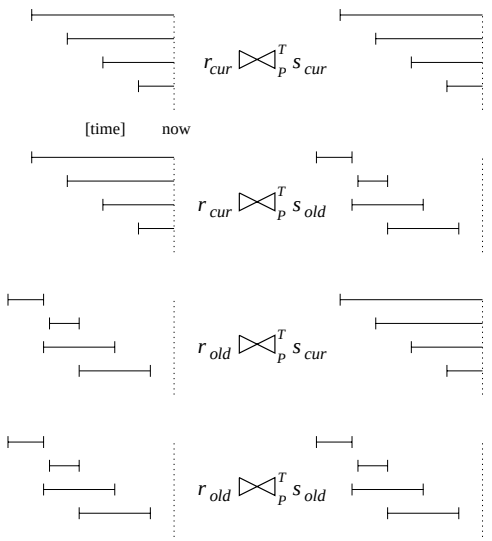


Figure 3: Temporal Join for Partitioned Storage

In the pseudo-code for the algorithms that follows, "**BNL loop**" denotes the basic loop in a block-based nested loop join [ME92, Dat95], with the additional property that conditions may be added that express the skipping of tuples in the reading of the relations. We use this abstraction to concisely convey the principles at play in the actual Java implementation that is used in the simulation studies, to be described in Section 4.

BNL, $r_{cur} \bowtie_P^T s_{cur}$

Two tuples qualify for the join result if they satisfy the join predicate P and if their time intervals overlap. When testing two tuples from the current partitions, the overlap test is omitted.

bnl{
allocate
    rel_{outer} = smaller relation;
    rel_{inner} = larger relation;
    $buffer_{outer}$ = $\max$;
    $buffer_{inner}$ = 1;
BNL loop
    if $P(t_{outer}, t_{inner})$ **then** add to result;
END; }

As is customary, the smaller relation is the outer relation. We assume a buffer of size $\max + 2$ blocks. One block is used for output, one is used for the inner relation, and the remainder is used for the outer relation. Tuple variables t_{inner} and t_{outer} range over the inner and outer relations, respectively.

TupleSkip, $r_{cur} \bowtie_P^T s_{old}$**,** $r_{old} \bowtie_P^T s_{cur}$

The algorithm computing the join of a current partition with an old partition exploits the ordering of the current partition on its interval start time $\mathrm{T}_{cur}^{\vdash}$ and of the old partition on its end time $\mathrm{T}_{old}^{\dashv}$. The current partition is the outer relation and the old is the inner.

The algorithm uses an aggressive buffer allocation strategy and allocates the maximum buffer size for the inner relation. The reason is that the part of the old partition that is relevant for the join might fit in the

buffer, even if the whole relation does not. If, during the computation, further tuples of the inner relation have to be read, the algorithm allocates the maximum memory-size for the outer relation, thus reverting to the normal nested-loop behavior.

tupleSkip{
allocate
rel_{outer} = current relation;
rel_{inner} = old relation;
buffer$_{outer}$ = 1;
buffer$_{inner}$ = max;
BNL loop
if ($P(t_{outer}, t_{inner}) \wedge overlap(t_{outer}[\mathrm{T}], t_{inner}[\mathrm{T}])$) **then** add to result;
if ($t_{outer}[\mathrm{T}^{\vdash}] > t_{inner}[\mathrm{T}^{\dashv}]$) **then**
skip rest of tuples in rel_{inner}; (*proceed with next $t \in rel_{outer}$*)
if ($size$(tuples read) $>$ *buffer*$_{inner}$) **then**
allocate
buffer$_{outer}$ = max;
buffer$_{inner}$ = 1;
END; }

For each tuple in the outer relation, the algorithm scans the inner relation for matching tuples. As soon as $\mathrm{T}^{\vdash}_{cur} > T^{\dashv}_{old}$ is true, the scan continues at the beginning of the inner relation for the next tuple in the outer relation. If for the last tuple in the outer relation, the above condition is true, the algorithm stops.

Figure 4 and Table 2 exemplify the TupleSkip join. Table 2 shows the reduction of the join load with this alogrithm. Tuple pairs that are inspected are marked by the letter "i", and pairs that join are marked by the letter "j". Unmarked pairs that are neither inspected nor join represent the reduction over a nested loop join.

Sales	2	6 - now
Marketing	5	7 - now
Controlling	4	7 - now

$\bowtie^{T}_{P}$

Bill	Marketing	2 - 5
Dana	Sales	4 - 6

Figure 4: $\mathrm{deptLocation}_{cur} \bowtie^{T}_{P} \mathrm{empDepartment}_{old}$

$\mathrm{deptLocation}_{cur}[\mathrm{T}]$	$\mathrm{empDepartment}_{old}[\mathrm{T}]$	
	2 - 5	4 - 6
6 - now	i	i, j
7 - now		i
7 - now		i

Table 2: Reduction of Join Load with TupleSkip

BlockSkip, $r_{old} \bowtie^{T}_{P} s_{old}$

The algorithm to compute the join of two old partitions is based on the ordering of the relations on $\mathrm{T}^{\dashv}$. The maximum buffer is allocated for the smaller outer relation. Now, for each block in the outer relation, we read through the inner relation. The algorithm proceeds with the next block of the outer relation if a tuple

read from the inner relation preceeds *all* tuples currently in the buffer from the outer relation. Here the algorithm differs from the previous one, for which the tuples in the outer relation are ordered on $\mathrm{T}^{\vdash}$, and the value of $\mathrm{T}^{\dashv}$ is constant, i.e., *now*.

blockSkip{
allocate
rel_{outer} = smaller relation;
rel_{inner} = larger relation;
buffer$_{outer}$ = max;
buffer$_{inner}$ = 1;
BNL loop
if $(P(t_{outer}, t_{inner}) \ \wedge \ overlap(t_{outer}[\mathrm{T}], t_{inner}[\mathrm{T}]))$ **then** add to result;
if $(\forall t_{outer} \in block_{outer}(t_{outer}[\mathrm{T}^{\vdash}] > t_{inner}[\mathrm{T}^{\dashv}]))$ **then**
skip rest of tuples in rel_{inner}; (*read next block of rel_{outer}*)
END; }

Similar to for the TupleSkip join, Figure 5 and Table 3 exemplify the BlockSkip join. Table 3 shows the reduction of the join load with this alogrithm. The block size for the outer relation is two tuples, and the i's and j's have the same meaning as before.

Bill	Marketing	2 - 5
Dana	Sales	4 - 6

$\bowtie_P^T$

Controlling	1	1 - 2
Controlling	2	2 - 3
Sales	5	3 - 5
Marketing	1	4 - 6

Figure 5: $\mathrm{empDepartment}_{old} \bowtie_P^T \mathrm{deptLocation}_{old}$

$\mathrm{empDepartment}_{old}[\mathrm{T}]$	$\mathrm{deptLocation}_{old}[\mathrm{T}]$			
	1 - 2	2 - 3	3 - 5	4 - 6
2 - 5		i	i	i, j
4 - 6		i	i, j	i

Table 3: Reduction of Join Load with BlockSkip

Together, the three algorithms described in this section allow us to compute the four sub-joins and thus the temporal join. We shall see next that the three algorithms also constitute the components of the incremental temporal join algorithm.

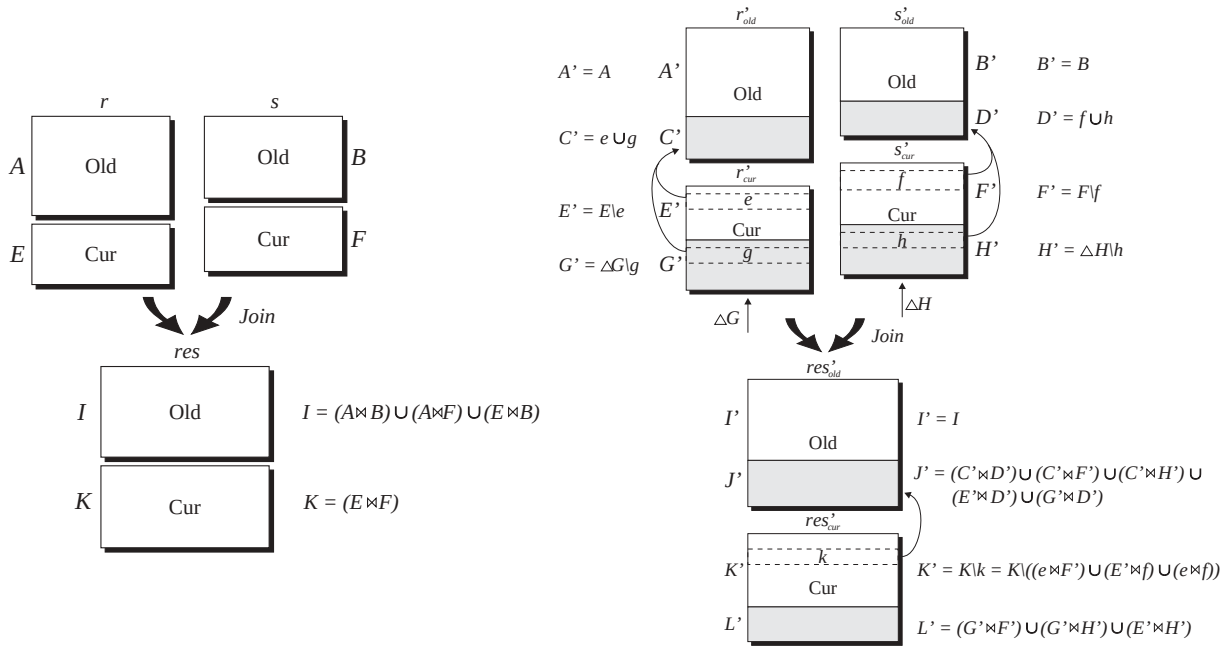
2.3 Incremental Join Computation

Computing a join incrementally is possible if the result of a previous computation of the same join is available. The incremental strategy is most attractive if the join is costly to recompute and if the changes to the underlying argument relations since the most recent computation are small. For example, this may apply to data warehouses, which are typically temporal and contain very large numbers of tuples, making join and computation expensive.

The goal when designing an incremental algorithm is to maximize reuse of the previously computed result. Since we do not physically delete tuples from our database, all the tuples of the outdated result will also appear in the newly computed one.

In the following, we describe the updates to an outdated join result that are necessary to obtain the up-to-date join result. Figure 6(a) shows the joins necessary to recompute the join result res of the argument relations r and s. The result res comprises the current partition K and the old partition I.

Figure 6(b) shows the operations necessary to incrementally compute the new up-to-date join result res' on the argument relations r' and s', by maximally reusing the old result, res. We first explain how the relations r and s evolve into r' and s', then explain the computation of res'.



(a) partitioned temporal join at t_0 (b) incremental join at t_1

Figure 6: Overview Incremental Join Computation

As time progresses, new tuples are added to the relations r and s. The shaded parts of the boxes representing the relations in Figure 6(b) show the changes between the two computations (at times t_0 and t_1). For relation r', we see that the tuple sets e and g are added to the old partition. The set e contains the tuples that were current at t_0, but were logically deleted between t_0 and t_1. The set g contains the tuples that were inserted after t_0, and logically deleted between t_0 and t_1. Furthermore tuple set G' is added to, and set e is deleted from, the current partition. Set G' contains the tuples that were inserted after t_0 and remain current at t_1. The changes described above apply similarly to s'.

Considering now the composition of the result res', similar explanations apply. The result res' comprises the current parts K' and L', as well as the outdated parts I' and J'. Part K' represents the part of the result already computed at t_0, which is K minus the set of tuples k. This set consists of all the tuples in K that derived from the sets e and f of now outdated tuples. Part L' derives from the newly added tuples in the argument relations. Part I' is identical to I, the old partition of the outdated result. Part J' derives from tuples logically deleted after t_0.

Below, we derive the expression to compute res', reusing the result res. The naming of relations and parts of relations is as depicted in Figure 6. The partioned join of two relations r' and s' comprises four individual joins and can be written as follows.

$$\begin{aligned} r' \bowtie_P^T s' &= (r'_{old} \cup r'_{cur}) \bowtie_P^T (s'_{old} \cup s'_{cur}) \\ &= \underbrace{(r'_{old} \bowtie_P^T s'_{old}) \cup (r'_{old} \bowtie_P^T s'_{cur}) \cup (r'_{cur} \bowtie_P^T s'_{old})}_{res'_{old}} \cup \underbrace{(r'_{cur} \bowtie_P^T s'_{cur})}_{res'_{cur}} \end{aligned}$$

The objective is to transform this expression for *res* into an equivalent expression that maximally reuses the components of the outdated result res'. To separate these preexisting components from the updated parts, we substitute $A' \cup C'$ for r'_{old}, $E' \cup G'$ for r'_{cur}, $B' \cup D'$ for s'_{old}, and $F' \cup H'$ for s'_{cur} (cf. Figure 6(b)). With each argument now consisting of four partitions, we obtain sixteen joins. These are shown in the first "column" in the derivation given next. The second "column," between the braces, gives equivalent reduced expressions. The expressions after the braces to the left (in the third "column") provide some of the key properties used in deriving the reduced expressions.

$$
\begin{array}{rcll}
r' \bowtie^T_P s' & = & \left.\begin{array}{l}(A' \bowtie^T_P B')\cup\end{array}\right\} & = A \bowtie^T_P B \left\{ \begin{array}{l} A' = A, B' = B \end{array}\right. \\
 & & \left.\begin{array}{l}(A' \bowtie^T_P (f \cup h))\cup \\ (A' \bowtie^T_P F')\cup \\ (A' \bowtie^T_P H')\cup\end{array}\right\} & = A \bowtie^T_P F \left\{ \begin{array}{l} F = F' \cup f \\ (A' \bowtie^T_P h) \cup (A' \bowtie^T_P H') = \emptyset \end{array}\right. \\
 & & \left.\begin{array}{l}((e \cup g) \bowtie^T_P B')\cup \\ (E' \bowtie^T_P B')\cup \\ (G' \bowtie^T_P B')\cup\end{array}\right\} & = E \bowtie^T_P B \left\{ \begin{array}{l} E = E' \cup e \\ (g \bowtie^T_P B') \cup (G' \bowtie^T_P B') = \emptyset \end{array}\right. \\
 & & \begin{array}{l}(C' \bowtie^T_P D')\cup\end{array} & \\
 & & \left.\begin{array}{l}(C' \bowtie^T_P F')\cup \\ (C' \bowtie^T_P H')\cup\end{array}\right\} & = C' \bowtie^T_P s'_{cur} \\
 & & \left.\begin{array}{l}(E' \bowtie^T_P D')\cup \\ (G' \bowtie^T_P D')\cup\end{array}\right\} & = r'_{cur} \bowtie^T_P D' \\
\multicolumn{4}{c}{\underbrace{\hspace{30em}}_{res'_{old}}} \\
 & & \left.\begin{array}{l}(E' \bowtie^T_P F')\cup\end{array}\right\} & = (E \bowtie^T_P F) \setminus k \left\{ \begin{array}{l} E = E' \cup e, F = F' \cup f \\ k = (e \bowtie^T_P F') \cup (E' \bowtie^T_P f) \cup (e \bowtie^T_P f) \end{array}\right. \\
 & & \begin{array}{l}(E' \bowtie^T_P H')\cup\end{array} & \\
 & & \left.\begin{array}{l}(G' \bowtie^T_P F')\cup \\ (G' \bowtie^T_P H')\end{array}\right\} & = G' \bowtie^T_P s_{cur} \\
\multicolumn{4}{c}{\underbrace{\hspace{30em}}_{res'_{cur}}}
\end{array}
$$

Using the reduced expressions above, six joins compute the updated old partition, res'_{old}, and three joins compute the updated current partition, res'_{cur}. In the derivation below, we isolate the parts from these partitions already available from the outdated result, res.

The three joins that form part I' of res'_{old} are already available as part I. The remaining three joins in part J' must be computed. The current partition res'_{cur} is composed of the parts K' and L'. Part K' is contained in the already available $K = E \bowtie^T_P F$. To reuse K, we derive K' by subtracting the components of part k from K. For part L' we need to perform two join operations.

$$
\begin{aligned}
r' \bowtie_P^T s' \;=\; & \underbrace{\left.\begin{array}{l}(A \bowtie_P^T B) \cup \\ (A \bowtie_P^T F) \cup \\ (E \bowtie_P^T B) \cup\end{array}\right\} = I = I'}_{} \\
& \left.\begin{array}{l}(C' \bowtie_P^T D') \cup \\ (C' \bowtie_P^T s'_{cur}) \cup \\ (r'_{cur} \bowtie_P^T D') \cup\end{array}\right\} = J' \\
& \underbrace{\qquad\qquad\qquad\qquad}_{res'_{old}} \\
& \left.\begin{array}{l}(E \bowtie_P^T F) \setminus k \cup\end{array}\right\} = K' = K \setminus e \bowtie_P^T F' \setminus E' \bowtie_P^T f \setminus e \bowtie_P^T f \\
& \left.\begin{array}{l}(E' \bowtie_P^T H') \cup \\ (G' \bowtie_P^T s_{cur})\end{array}\right\} = L' \\
& \underbrace{\qquad\qquad\qquad\qquad}_{res'_{cur}}
\end{aligned}
$$

To incrementally compute the temporal join $r' \bowtie_P^T s'$, we thus need to perform a total of eight joins. At first sight, this might seem to be no improvement over the four joins necessary to recompute the result from scratch. However, in the incremental computation, we reuse the old result, slightly updating it. Depending on the outdatedness of the available outset, each of the eight joins will involve at most one large relation. Such joins are efficient to compute.

Next, for the computation of the eight joins, we need not read the entire stored relations r and s', but parts of them. Relations r and s are updated to r' and s', respectively, in such a way that all parts used in the incremental computation, namely C', D', r_{cur}, s_{cur}, E', F', G', and H' are contained in blocks of tuples newly added to the stored relations. Thus, if we know the time t_0 at which the outdated result res was computed (which we do), we can obtain the relations necessary to compute the new result res' as parts of stored relations by using conditional read operations. The tuples of e and f are mixed into the E' and F' blocks in the relations r'_{cur} and s'_{cur}.

For the computation of the eight joins, we make use of the algorithms described in the previous section. Below we show a simplified algorithm for the incremental computation of a temporal join. The function *subtract*(set1, set2) deletes all elements from set1 that also occur in set2, whereas *add*(set1, set2) appends all elements of set2 to set1. The *subtract* operation can be seen as an additional join operation, for which the result consists of tuples in set1, but not in set2.

incr{
add(I, BlockSkip($C' \bowtie_P^T D'$));
add(I, TupleSkip($r_{cur} \bowtie_P^T D'$));
add(I, TupleSkip($s_{cur} \bowtie_P^T C'$));
subtract(K, TupleSkip($e \bowtie_P^T F'$));
subtract(K, BNL($E' \bowtie_P^T f$));
subtract(K, BNL($e \bowtie_P^T f$));
add(K, BNL($G' \bowtie_P^T s_{cur}$));
add(K, BNL($E' \bowtie_P^T H'$)); }

Having completed the design of the recomputation and incremental temporal join algorithms, the next step is to gain an understanding of their performance characteristics.

3 Analytical Cost Formulas

This section presents formulas for estimating the costs of the algorithms presented in the previous section. Specifically, the two following subsections give formulas for the cases of recomputation and incremental

computation. First, some general assumptions are made.

In general, the cost of the join $r \bowtie_P^T s$ consists of the cost of input/output (IO) operations, C_{IO}, plus the CPU cost, C_{CPU}. We focus on the IO cost and omit for simplicity the CPU cost. Next, the IO cost includes the cost of read (R) and write (W) operations, C_R and C_W, respectively. Again for simplicity, we do not distinguish between sequential and random IO operations. We expect most IO operations to be sequential for all the algorithms. The cost C_W for writing to disk is typically assumed to be identical for algorithms computing the same results and is thus frequently ignored when comparing the costs of different join algorithms. But when comparing recomputation and incremental computation, this assumption does not hold, and we consequently consider this cost.

3.1 Recomputation

We give formulas for C_R, the disk read cost of the temporal join algorithms, based on data characteristics including tuple lifespans and relation lifespans. The *tuple lifespan* of a tuple is the duration of the tuple's time interval. The *relation lifespan* of a relation is the duration of the interval from the earliest start time of a tuple in the relation to the latest end time of a tuple in the relation.

We assume tuples in the old partition of a relation have the same (standard) lifespan and also the lifespans to be *uniformly distributed* over the lifespan of the partition. For the current partition of a relation, where tuples end at the special chronon *now* and thus are still growing, these assumptions imply that there are as many tuples inserted as there are tuples deleted, and that the lifespan of the current partition is identical to the standard tuple lifespan.

With these assumptions, we can develop a precise analytical cost formula that will serve as a good approximation for more general cases. For example, the standard lifespan may represent well a situation with an average lifespan and tuples randomly distributed over the relation.

The cost of a temporal join for partitioned storage is the sum of the costs of the four individual joins, $r_{cur} \bowtie_P^T s_{cur}$, $r_{cur} \bowtie_P^T s_{old}$, $r_{old} \bowtie_P^T s_{cur}$, and $r_{old} \bowtie_P^T s_{old}$.

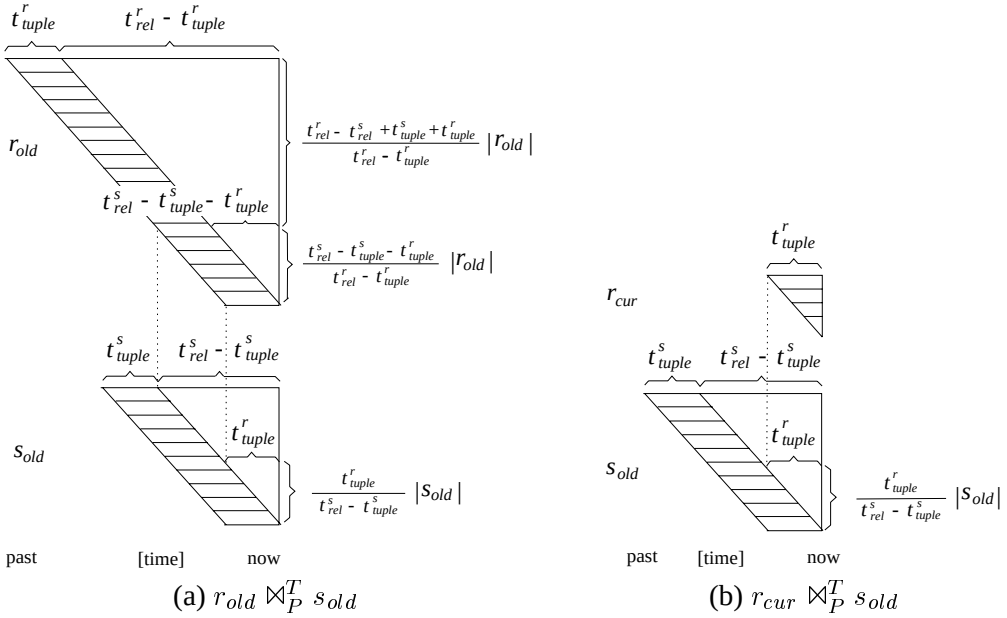
The following formulas estimate the tuple reads C_r in blocks, where m is the size of the main-memory buffer in blocks, and $|r|$ is the size of relation r in blocks. The functions sel1(rel_{old}, rel_{old}) and sel2(rel_{cur}, rel_{old}) represent the selectivity of the BlockSkip and the TupleSkip algorithm, respectively.

$$
\begin{aligned}
C_R \;=\; & |r_{cur}| + \frac{|r_{cur}|}{m}|s_{cur}| + && (1)\\
& |r_{cur}| + \frac{|r_{cur}|}{m}|s_{old}| \cdot \mathrm{sel2}(r_{cur}, s_{old}) + && (2)\\
& |r_{old}| + \frac{|r_{old}|}{m}|s_{cur}| \cdot \mathrm{sel2}(s_{cur}, r_{old}) + && (3)\\
& |r_{old}| + \frac{|r_{old}|}{m}|s_{old}| \cdot \mathrm{sel1}(r_{old}, s_{old}) && (4)
\end{aligned}
$$

The cost of a partitioned computation without the selectivity factors is in the range of $|r|$ to $|r| + |s|$ higher than the cost of the regular nested loop computation without partitioned storage because we have to perform four joins instead of one and thus also have to read each partition of the outer relation twice. The cost is $|r|$ higher in the case none of the relations completely fit in the buffer, and is $|r| + |s|$ higher in the case a relation (r or s) fits entirely. However, by exploiting the orderedness properties of the relations, we can reduce the costs of three of the four joins. This reduction is expressed by the selectivity factors sel1 and sel2 in the cost formula.

The next step is to estimate the selectiviy factors for the BlockSkip and TupleSkip algorithms. Figure 7 gives graphical illustrations of the situations for these two algorithms. The relations at the top of the figure

are the outer relations in the join algorithms. In our case, those are the relations r_{old} and r_{cur}. The inner relations in the inner loop, in both cases s_{old}, are scanned sequentially for each tuple in the outer loop.



(a) $r_{old} \bowtie^T_P s_{old}$ (b) $r_{cur} \bowtie^T_P s_{old}$

Figure 7: Temporal Joins Using (a) BlockSkip and (b) TupleSkip

The derivation of the cost formulas is based on proportions using similar triangles. The sides of the triangles we relate to each other are, horizontally, a time interval, and, vertically, a measure for the number of tuples. We use these proportions to illustrate the correspondence between time and number of tuples, i.e., for a given time interval *I*, starting at *now* and reaching *x* chronons into the past, we want to know how many time intervals of the relation, and thus with how many tuples, *I* overlaps with. In Appendix A.1 we derive a formula that computes the number of tuples that start before a given time point. Consider now the time interval t^r_{tuple} in the relation s_{old} of Figure 7(a). This interval overlaps with $(t^r_{tuple})/(t^s_{rel} - t^s_{tuple}) \cdot |s_{old}|$ tuples of s_{old}. By using these proportions, we derive in the following sections and in Appendix A cost formulas for the selectivity factors sel1 and sel2.

3.1.1 sel1, BlockSkip

Figure 7(a) shows two relations r_{old} and s_{old} to be joined. For each tuple in r_{old}, all tuples from s_{old} that satisfy $T^{\dashv}_s > T^{\vdash}_r$ have to be read. In Figure 7(a), a dotted line shows this condition for the newest tuple in r_{old}. Once a tuple from s_{old} is read that does not satisfy this condition, the remainder of s_{old} can be skipped for the tuple in r_{old}. The cost of the algorithm is given below and is derived in Appendix A.2. For relations r_{old} and s_{old}, we denote the lifespans of the relations by t^r_{rel} and t^s_{rel}, and the tuple lifespans by t^r_{tuple} and t^s_{tuple}, respectively.

$$\text{sel1} = 0.5 \cdot (1 + \frac{t^r_{tuple}}{t^s_{rel} - t^s_{tuple}}) \cdot \frac{t^s_{rel} - t^s_{tuple} - t^r_{tuple}}{t^r_{rel} - t^r_{tuple}} + \frac{t^r_{rel} - t^s_{rel} + t^s_{tuple} + t^r_{tuple}}{t^r_{rel} - t^r_{tuple}} \tag{5}$$

The formula is the sum of two parts. The first quantifies the selectivity for the tuples 1 to $(t^s_{rel} - t^s_{tuple} - t^r_{tuple})$ in r_{old}. The last tuple is the first for which we have to read all tuples in s_{old}. In Figure 7(a), a dashed line shows the link between the end of the last tuple in s_{old} and the beginning of the first overlapping tuple in r_{old}. The second part of Formula 5 computes the "selectivity" of the remaining tuples in r_{old}, for which we have to read all tuples in s_{old}.

Assuming that both relations have identical tuple lifespans, $t^r_{tuple} = t^s_{tuple} = t_{tuple}$, and that $t_{tuple} \ll t^s_{rel}$, the expression for sel1 can be simplified to the following.

$$\text{sel1} = \frac{t^r_{rel} + 2t_{tuple} - 0.5 \cdot t^s_{rel}}{t^r_{rel} - t_{tuple}} \tag{6}$$

As an example, assume $t^r_{rel} = t^s_{rel} = 100$ chronons and $t_{tuple} = 1$ chronon. These numbers mean that the relations have equal lifespans and the tuple lifespan is small compared to the relation lifespan. In this case, sel1 approaches 0.5. In general, the shorter the tuple lifespan t_{tuple} compared to the relation lifespan t^r_{rel}, the smaller is sel1 and thus the cost for computing $r_{old} \bowtie^T_P s_{old}$.

3.1.2 sel2, TupleSkip

In the following we give the selectivity for the join of a current partition r_{cur} and an old partition s_{old} using the TupleSkip algorithm (cf. Figure 7(b)). For the old partition s_{old}, we denote the lifespan of the relations by t^s_{rel}, and the lifespan of a tuple by t^s_{tuple}. In the case of the current relation, however, the tuple lifespan equals the relation lifespan, denoted by t^r. The formula below is derived in Appendix A.3.

$$\text{sel2} = 0.5 \cdot \frac{t^r}{t^s_{rel} - t^s_{tuple}} \tag{7}$$

To exemplify, let $t^r = t^s_{tuple} = 10$ chronons, and $t^s_{rel} = 100$ chronons. These numbers mean that the tuple life spans are one tenth of the relation lifespan. In this case $\text{sel2} = \frac{1}{18}$. In general, the smaller t^r in relation to t^s_{rel}, the smaller is sel2. In the extreme case, sel2 approaches values close to 0.

3.2 Incremental Computation

The costs of reading C_R and writing C_W tuples for the incremental join algorithm (Section 2.3) stem from the the costs associated with the computations of the eight constituent joins, in addition to the costs of adding and subtracting these join results to and from the stored relations.

The incremental join algorithm reuses the sub-join algorithms from the recomputation algorithm that we considered in the previous section. For all eight joins, at least one of the joining relations is expected to be small, thus yielding a relatively low cost of computing all eight joins. The cost of the add operations is simply that of writing the tuples to file. The incremental algorithm also incorporates the deletion of tuples (part k) from the current partition of the old result (part K). This deletion can be computed as a join with a predicate that returns tuples that are in K, but not in k.

The total cost of the incremental computation is the cost of the eight sub-joins plus the cost of reading and writing for adding and subtracting the results of those joins (cf. Section 3.1).

4 Performance Study

This section first explains the overall design and objectives of the study, including data generation. It then proceeds to compare the recomputation algorithms and finally compares recomputation with incremental computation. A summary of the findings is included at the end.

4.1 General Considerations

Using the implementations of the join algorithms described earlier in the paper, this section reports on simulation-based experiments aiming at understanding the performance characteristics of the proposed algorithms.

The studies aim to obtain insight on a total of three aspects. First, it is of interst to understand how the performance of the nested-loop (NL) versus the sort-merge-based (SMB) joins relate for varying main-memory sizes. Second, the characteristics of the NL and SMB joins for varying kinds of argument data are of interest. In particular, it is of interest to learn for what kinds of data, the NL join outperforms the SMB join and vice versa. Third, it is relevant to learn under what circumstances recomputation outperforms incremental computation, and vice versa.

As the performance measure, we use the number of input/output (IO) operations. The read operations encompass random as well as sequential reads, with random reads weighted with factor 10. For the comparisons of recomputation algorithms, such as the NL and our SMB algorithm, we do not consider write operations. However, when comparing incremental computation with recomputation, the number of write operations will differ among algorithms and are thus included in the performance measure.

Parameter	**Unit**	**Standard**	**Other Values**	**Comments**
Relation size	tuples	20,000		
Relation lifespan	chronons	75,000		
Distribution of intervals		uniform		
Buffer size	fraction of total relation size	1/16	1/1, 1/2, 1/4, 1/8, 1/32, 1/64	
Tuple lifespan	chronons	1,600	2x, 4x, 8x, 16x	multiples of a single lifespan
Number of long lived tuples	% of tuples	0	10, 20, 30, 40, 60, 80	
Outdatedness of old result used for inc. comp.	chronons from *now*	0	5, 15, 25, 35, 45, 55, 65	in thousands of chronons

Table 4: Performance Study Parameters

The simulations in the study use different settings for various parameters, including *main-memory size* and *data characteristics*. The data characteristics considered include the *percentage of long-lived tuples* and the *tuple length*, both of which affect the selectivity and thus the cost of a temporal join. Table 4 presents the parameters, their units of measurement, and their standard settings. The first three parameters are fixed throughout the performance studies at their standard values. For the remaining parameters, the standard value is used unless explicitly stated otherwise. For these parameters, the values beyond the standard settings are also reported, and explanatory comments are offered where appropriate.

To keep the experiments manageable while still obtaining realistic results, we use relatively small relations of size 20,000 tuples, but then compensate by also assuming a small block size, where one block corresponds to one tuple. Following these decisions, all sizes are reported in numbers of tuples.

For the experiments we generate data using the TimeIT software [KS98]. TimeIT is a system for testing temporal database algorithms, and it contains a database generator that generates interval timestamped temporal relations. Both the positions of timestamps within the lifespan of a relation, as well as the duration of the timestamps can be selected from several distributions, including uniform, normal, constant, and percentage breakdowns. As an example of the latter, 25% of the timestamps' start times may be determined by a uniform distribution between 1000 and 10000 chronons, and 75% might then be normal distributed with 5000 chronons as the mean; the durations of the tuples would be specified by separate distributions. Explicit attributes may be specified with similar distributions.

4.2 Comparing Recomputation Algorithms

In this section, we compare the SMB algorithm to a version of its existing competitor, the nested-loop (NL) join. The NL algorithm is not based on partitioned storage, thus we do not impose the cost of reading partitioned relations onto the algorithm. These experiments should furthermore show under what circumstances a partition-based algorithm (SMB) can outperform a non-partitioned competitor (NL). We compare the algorithms under varying parameter settings, specifically, using varying main memory buffer sizes, varying percentages of long-lived tuple timestamps, and varying timestamp lengths.

4.2.1 Sensitivity to Main Memory Buffer Size

An important factor for join performance is the size of the main memory available for the join. In the present experiment we compare the NL and SMB joins under varying main-memory buffer sizes. The buffer sizes are specified in fractions of the size of one relation. We use 1/1, 1/2, 1/4, 1/8, 1/16, and 1/32 as main memory buffer in our experiments. All other parameters assume their standard values, as shown in Table 4. Figure 8 presents the results. The experiments show that the SMB join yields better performance for small

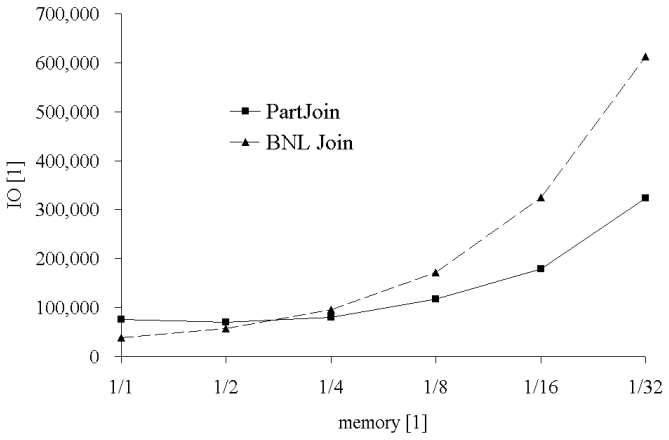
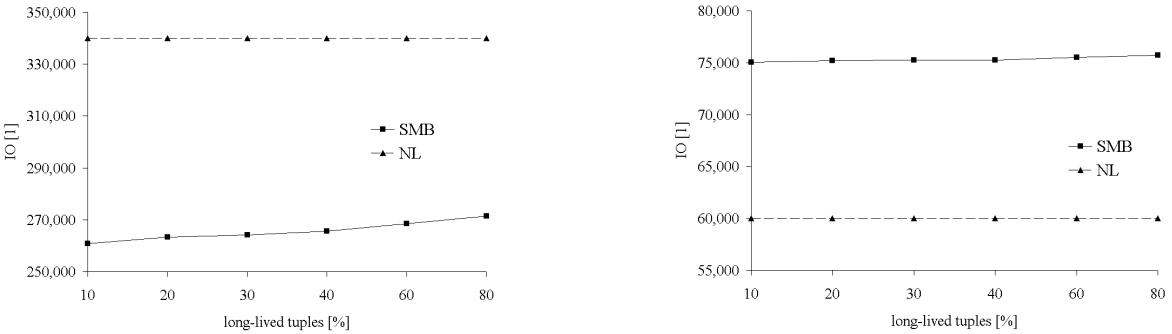


Figure 8: NL Versus SMB Join for Varying Buffer Sizes

main-memory sizes. In this case, the SMB join's reduction criteria are successful in reducing the cost of computation. However, in the case one relation fitting entirely in memory, the NL join performs better due to the additional reading cost for a join in a partitioned storage environment (cf. Section 3.1).

4.2.2 Effects of Long-Lived Tuples

An aspect of data that typically affects the performance of a temporal join is the fraction of tuples with an untypically long interval timestamp. For our experiments, we choose a duration of 10 times the average of standard tuples for long-lived tuples. Figures 9(a) and (b) show the performance of the NL and SMB join under varying percentages (10% to 80%) of long-lived tuples. In addition, we conducted these experiments with two different buffer sizes. The results show that the performance of the SMB algorithm degrades with increasing percentage of long-lived tuples, whereas the NL algorithm remains unaffected. The effect of long-lived tuples on the degradation of the join performance seems weaker in the case of large buffer sizes. Increasing the tuple lifespan means that the algorithm is forced to read more tuples. The IO cost associated with this becomes relatively smaller as the buffer size increases. Thus, with a large buffer available, an increased number of long-lived tuples has a much smaller effect on the join performance.



(a) buffer size 1/16 of relation size (b) buffer size 1/2 of relation size

Figure 9: NL Versus SMB Join for Varying Percentages of Long-Lived Tuples

4.2.3 Effects of Varying Tuple Lifespans

In the previous section, we varied the number of long-lived tuples relative to the total number of tuples in the relation. Another approach to affect the temporal characteristics of the argument temporal databases is to vary the lifespan of all tuples. The results obtained when doing this are shown in Figure 10. It can be

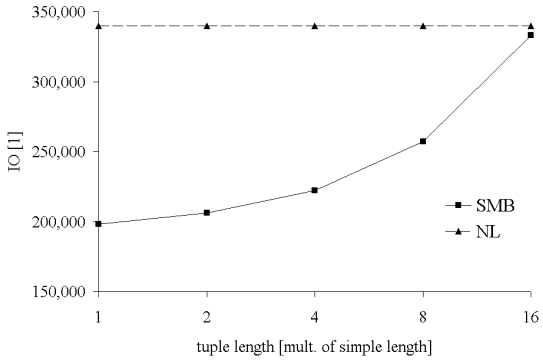


Figure 10: NL Versus SMB Join for Varing Tuple Lifespans

seen that the performance of the SMB join degrades with increasing tuple lifespan. This result matches the analytical studies in Section 3 that show that the selectivity factors sel1 and sel2 approach 1 as the tuple lifespan increases. In the case that the number of outdated tuples is rather large compared to the number of current tuples, the cost of the whole join operation is mostly determined by the BlockSkip algorithm. Thus, if the selectivity factor for this join, sel1, converges to 1, the cost of the whole join converges to the cost of the equivalent NL join.

4.3 Incremental Computation Versus Recomputation

The experiments reported here aim to explore the break-even point between the incremental and SMB joins. The degree of outdatedness of the outset for an incremental join fundamentally affects the relative performance of the two. We adopt the outdatedness of the old result as the parameter that we vary in the experiments. We assume that the incremental join (and the recomputation join) take place at the current time. In Figure 11, the x-axis indicates the outdatedness of the outdated result by giving the time at which the outdated result was computed in numbers of chronons before the current time where the incremental

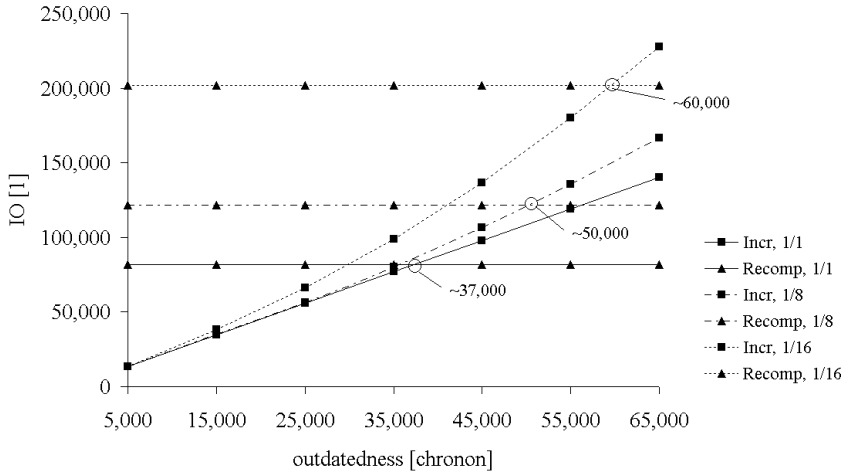


Figure 11: Recomputation Versus Incremental Computation using Varing Outdatedness and Buffer Sizes

computation is performed. We conducted our experiments for buffer sizes of 1/1, 1/8, and 1/16 of the relation size.

In the performance measurements, we encountered the situation that the outdated current partition (K) of the result (cf. the *subtract*() operation in Section 2.3) did not contain any current tuples, and thus was completely moved to the old partition of the result.

The break-even point between incremental computation and recomputation in Figure 11 is at about 37,000, 50,000, and 60,000 chronons for the buffer sizes of 1/1, 1/8, and 1/16, respectively. This means when an old result was computed at a time corresponding to chronon 38,000, 25,000, and 15,000, respectively, or later, incremental computation is better than recomputation.

Viewing these results in the light of the experiments in Section 4.2.1, one would expect incremental computation to always be better than recomputation. This is not always the case as these experiments show. To incrementally compute a join we need to compute eight individual joins (Section 2.3). The results of those eight joins need to be added to or subtracted from existing relations. The cost of computing these joins and the cost of adding and subtracting the respective results can be higher than the cost of recomputation.

Increasing the buffer size also disfavors the incremental computation, since larger buffer size means generally lower cost of join computation (cf. Section 4.2.1).

4.4 Summary of Performance Study

The sort-merge based (SMB) algorithm outperforms the nested-loop (NL) algorithm, except when main memory is so large that an entire relation fits in memory.

The temporal relation parameters, tuple lifespan and percentage of long-lived tuples generally have a smaller impact on the performance of the SMB algorithm than does the main-memory buffer size.

We compared the performance of the SMB algorithm to its incremental version, varying both the outdatedness of the argument join result in the incremental computation and the buffer size. The studies favor the incremental algorithm for the cases of low to modest outdateness. The degree of outdatedness necessary to competitively perform an incremental computation varies with the main memory size. The smaller the buffer size, the more outdated a result can be while incremental computation being superior to recomputation. Generally, the results suggest that incremental computation may be applied in many situations where recomputation would be a waste of resources.

5 Conclusions and Future Work

The paper formally defines a temporal join of two temporal relations and extends this definition to apply also to a partitioned storage environment. The paper then proceeds to define and study the characteristics of two new join algorithms for temporal relations with append-only characteristics, a sort-merge based algorithm and its incremental version.

The algorithms assume that the relations have associated an interval-valued time attribute. Beyond this distinguished attribute, no assumptions about the numbers of other attributes and their domains are made. The join predicate is the conjunction of an overlap predicate on the time attribute values and an arbitrary predicate on the remaining attributes. The algorithms work in a partitioned-storage environment, which is realistic for very large relations. That is, current and outdated tuples of a temporal relation are stored separately in a current and an old partition, respectively.

The paper includes analytical cost formulas for the joins and also reports on simulation-based performance studies. The performance studies show the sort-based algorithm to be an improvement over the only existing join algorithm that contends with the same class of predicates, namely the nested-loop join. Only in the case of large buffer sizes is the nested-loop algorithm competitive. This is due to the additional reading cost for the join of partitioned relations (four sub-joins), as opposed to the nested-loop join of unpartitioned relations (one join). This indicates that the sort-based algorithm is an overall good replacement for the nested loop algorithm for the data considered in this paper.

The included evaluation of the performance of the incremental algorithm with respect to the recomputation algorithm show the incremental algorithm to be superior when the available outset for the computation is outdated to a low or modest degree. The maximum degree of outdatedness possible, while still having the incremental algorithm be competitive, grows with decreasing main memory size. While incremental computation techniques have proven competitive in many settings, they also introduce a space overhead in the form of differential files. For the temporal data explored here, however, this overhead is avoided because the differential files are already part of the database.

This research points to several directions for future research. When performing incremental computation, previous join results must be cached for future use. Assuming that only limited disk space is available for caching, caching should be selective. Additional research in caching policies and cache replacement policies is warranted. Next, spatiotemporal data in many cases arrive at the database in a time-ordered fashion, thus meeting the assumptions made in this paper. Extending the join algorithms proposed here, or devising entirely new algorithms, is a relevant and interesting direction. The lack of good spatiotemporal indices adds to the relevance of this direction. Finally, the result of an incremental computation is sorted if it is cached for use in a later join computation. The optimal integration of this sorting in the algorithms remains to be explored.

Acknowledgements

The authors are with Department of Computer Science, Aalborg University, Fredrik Bajers Vej 7E, DK–9220 Aalborg Ø, DENMARK, {pfoser|csj}@cs.auc.dk. This research was supported in part by the Danish Technical Research Council through grant 9700780, by the CHOROCHRONOS project, funded by the European Commission, contract no. FMRX-CT96-0056, and by the Nykredit Corporation.

A Analytical Formulas

In this appendix, we derive the selectivity factors sel1 and sel2 used in the analytical cost formulas for the BlockSkip and the TupleSkip join algorithms. The formulas for sel1 and sel2 are shown in Formulas 5 and

7 in Section 3.1. To give a cost formula for the join of two relations r and s, we consider how many tuples in the respective relations can possibly join. We disregard the explicit join attribute and consider only the time interval of the tuples as the join criterion. Thus, two tuples join if their time intervals overlap. Making certain assumptions about the properties of the time intervals in the relations r and s, we can give estimates for how many tuples have to be read.

A.1 General Considerations

A first step is to develop a formula that computes the number of tuples that end after a given time point. This formula becomes useful for the cost formula of the temporal join $r \overset{T}{\bowtie}_{P} s$, for which we want to know how many tuples of s, we have to read for each tuple in r.

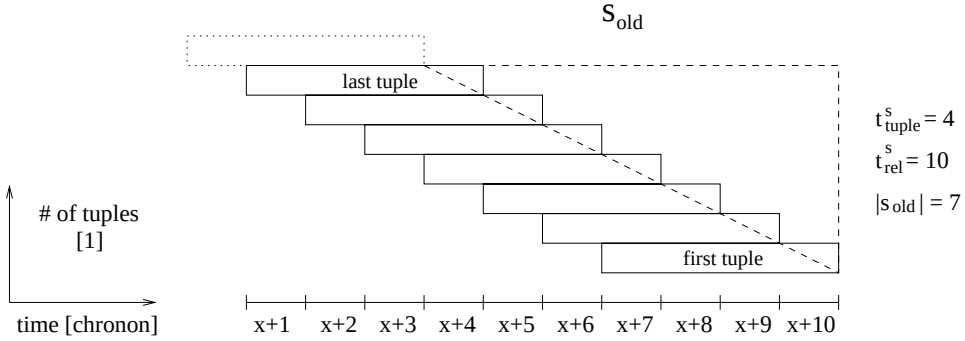


Figure 12: Relating Number of Tuples to Elapsed Time

Consider now Figure 12. Here we show the old partition s_{old} containing 9 outdated tuples. In the x-direction of the coordinate system, we depict the time relative to an arbitrary timepoint x. In the y-direction, we measure numbers of tuples. The tuples in Figure 12 are symbolized by bars of length 4. The length represents the tuple lifespan t^s_{tuple} in chronons. The lifespan of the whole relation is 10 chronons. In Figure 12, two tuples are labeled "first tuple" and "last tuple," respectively, to indicate which tuple we encounter first and last when traversing the relation.

The quantity we want to express is the sum of tuples per time unit. To do this, we make use of the dashed triangle in Figure 12. The height of the triangle is the number of tuples in s_{old}. The width is the time interval from x+10 to where the tuple preceding the last tuple (the dotted tuple in Figure 12) would end. This point is determined by subtracting the *spacing* between consecutive ends of time intervals from the *end of the last tuple*. To obtain a formula for the length of the spacing between end points, we have to distribute the end points of the $|s_{old}|$ time intervals, in our example 7, over the tuple lifespan t^s_{rel}, which is 10 (from x+1 to x+10). The last tuple starts at the beginning of the relation lifespan, at x+1. It ends at $\mathrm{x} + t^s_{tuple}$, in our example x+4. Thus, the time interval we have to distribute over our 7 ($|s_{old}|$) end points, is 6 chronons, from chronon x+5 to chronon x+10. Thus is computed as $t^s_{rel} - t^s_{tuple}$. Now, we have to distribute the 7 ($|s_{old}|$) end points over this interval, where the first end point shall be at the beginning of the interval and the last shall be at the end of it. This means we have 7−1 ($|s_{old}| - 1$) spaces between the end points. The length of the spacing can thus be computed by $(t^s_{rel} - t^s_{tuple})/(|s_{old}| - 1)$.

Effectively, the width of the triangle is computed by subtracting one tuple lifespan from the relation lifespan and adding the above derived spacing for tuple ends to it. This gives $t^s_{rel} - t^s_{tuple} + (t^s_{rel} - t^s_{tuple})/(|s_{old}| - 1)$. In our example, the width of the triangle would be 7.

By now, having both the height and the width of the triangle, we can express the quantity of tuples per

time unit.

$$\frac{\text{tuples}}{\text{time}} = \frac{|s|}{t^s_{rel} - t^s_{tuple} + \frac{t^s_{rel} - t^s_{tuple}}{|s|-1}} \tag{8}$$

In the example shown in Figure 12, this ratio is 1. Transforming Formula 8, we obtain the following much simpler expression.

$$\frac{\text{tuples}}{\text{time}} = \frac{|s| - 1}{t^s_{rel} - t^s_{tuple}} \approx \frac{|s|}{t^s_{rel} - t^s_{tuple}} \tag{9}$$

For the derivation of the cost formulas, we simplify the formula by using $|s|$ instead of $|s| - 1$ in the numerator. For a moderate number of tuples, this simplification will yield negligible differences from all prior formulas.

A.2 BlockSkip

In the case of the BlockSkip algorithm, two old partitions are joined. Figure 7(a) shows the time intervals of the relations r_{old} and s_{old}. For each tuple of r_{old} - the relation in the outer loop of the algorithm - we want to know how many tuples of s_{old}, we have to read. The cost of the whole join is then the sum of reads over all tuples in r_{old}.

At this point, note that the reasoning below only applies if t^r_{tuple}, the typical tuple lifespan of r_{old}, is shorter than $t^s_{rel} - t^s_{tuple}$. If this condition does not hold, it means that already for the first tuple in r_{old}, we have to read all the tuples in s_{old}. In this case, we have no additional selectivity, and sel1 becomes 1.

For the first tuple in r_{old} that has a lifespan of t^r_{tuple} we compute by using Formula 9 that we have to read $(t^r_{tuple})/(t^s_{rel} - t^s_{tuple}) \cdot |s_{old}|$ tuples from s_{old}. The dotted line in Figure 7 illustrates this situation.

Now, since t^r_{rel} is bigger than t^s_{rel}, we will reach a tuple in r_{old} that overlaps with the last tuple of s_{old}. This means that for this and all the following tuples in r_{old}, we have to read the entire relation s_{old}. This is the case for tuple number $(t^s_{rel} - t^s_{tuple} - t^r_{tuple})/(t^r_{rel} - t^r_{tuple}) \cdot |r_{old}|$ of r_{old}.

We split the formula to compute the total number of tuples, we have to read from s_{old}, into two parts: one for the tuples in r_{old} for which we do not have to read the entire relation s_{old}, and another for which we have to.

For the first part, we use the formula of an arithmetic series. Generally the sum s_n of n elements of an arithmetic series is $s_n = n \cdot (a_1 + a_n)/2$, where a_1 is the value of the first element and a_n is the value of the last element. Using the numbers derived for our case we get the following.

$$s^1_n = 0.5 \cdot \frac{t^s_{rel} - t^s_{tuple} - t^r_{tuple}}{t^r_{rel} - t^r_{tuple}} \cdot |r_{old}| \cdot (\frac{t^r_{tuple}}{t^s_{rel} - t^s_{tuple}} \cdot |s_{old}| + |s_{old}|) \tag{10}$$

For the second part, we have to read for each of the remaining tuples in r_{old} all the tuples in s_{old}.

$$s^2_n = \frac{t^r_{rel} - t^s_{rel} + t^s_{tuple} + t^r_{tuple}}{t^r_{rel} - t^r_{tuple}} \cdot |r_{old}| \cdot |s_{old}| \tag{11}$$

The total number of reads for s_{old} is the sum of s^1_n and s^2_n. By factoring out $|r_{old}| \cdot |s_{old}|$, we obtain Formula 5 for sel1, shown in Section 3.1.

A.3 TupleSkip

In the case of the TupleSkip algorithm, a current partition is joined with an old partition. Figure 7(a) shows the time intervals of the relations r_{cur} and s_{old}. The relation r_{cur} is the one in the outer loop of the join.

Thus, we have to read all its tuples once. We want to know, for each tuple of r_{cur}, how many tuples of s_{old}, we have to read.

For the first tuple in r_{cur}, we assume that its lifespan is 0. Thus, we do not have to read any tuples from s_{old} for it. Making this assumption simplifies the cost formula. For the last tuple in the current partition, we have to read $(t^r_{tuple})/(t^s_{rel} - t^s_{tuple}) \cdot |s_{old}|$ tuples from s_{old}.

By again using the above formula to compute the sum of elements of a arithmetic series, we obtain the following formula for the number of tuples, we have to read from s_{old}.

$$s_n = 0.5 \cdot |r_{cur}| \cdot (0 \cdot |s_{old}| + \frac{t^r}{t^s_{rel} - t^s_{tuple}} \cdot |s_{old}|) \tag{12}$$

By factoring out $|r_{cur}| \cdot |s_{old}|$, we obtain the Formula 7 for sel2, as shown in Section 3.1.

References

[AS88] I. Ahn and R. T. Snodgrass. Partitioned Storage for Temporal Databases. *Information Systems*, 13(4):369–391, 1988.

[CDI+97] J. Clifford, C. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the Semantics of "now" in Databases. *ACM Transactions on Database Systems*, 22(2):171–214, June 1997.

[Cop82] G. Copeland. What If Mass Storage Were Free? *IEEE Computer Magazine*, 15(7):27–35, July 1982.

[Dat95] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, 6th edition, 1995.

[GS91] H. Gunadhi and A. Segev. Query Processing Algorithms for Temporal Intersection Joins. In *Proceedings of the IEEE Conference on Data Engineering*, pages 336–344, Los Alamitos, CA, USA, April 1991.

[JS94] C. S. Jensen and R. T. Snodgrass. Temporal Specialization and Generalization. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):954–974, 1994.

[Kim96] R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, Inc., 1996.

[KD79] K. C. Kinsley and J. R. Driscoll. Dynamic Derived Relations Within the RAQUEL II DBMS. In *Proceedings of the 1979 ACM Annual Conference*, pages 69–80, October 1979.

[KS98] N. Kline and M. Soo. Time-IT, the Time-Integrated Testbed. URL: <ftp://ftp.cs.arizona.edu/timecenter/time-it-0.1.tar.gz>, Current as of August 18, 1998.

[LM92] T. Y. C. Leung and R. R. Muntz. Temporal Query Processing and Optimization in Multiprocessor Database Machines. In *Proceedings of the International Conference on Very Large Databases*, pages 383–394, Vancouver, Canada, August 1992.

[LM93] T. Y. C. Leung and R. R. Muntz. *Stream Processing: Temporal Query Processing and Optimization*, In A. U. Tansel et al. (editors), Temporal Databases: Theory, Design, and Implementation, Chapter 14, pages 329–355. Benjamin/Cummings, 1993.

[ME92] P. Mishra and M. H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[PJ98] Pedersen, T. B. and C. S. Jensen. Research Issues in Clinical Data Warehousing. In *Proceedings of the Tenth International Conference on Scientific and Statistical Database Management*, pages 43–52, July 1998. IEEE Computer Society.

[QW91] X-L. Qian and G. Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[Rou91] N. Roussopoulos. An Incremental Access Method for Viewcache: Concept, Algorithm, and Cost Analysis. *ACM Transactions on Database Systems*, 16(3):535–563, September 1991.

[SA85] R. T. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pages 236–246, Austin, TX, May 1985.

[Sno87] R. T. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*,12(2):247–298, June 1987.

[SS93] A. Segev and A. Shoshani. *A Temporal Data Model Based on Time Sequences*, In A. U. Tansel et al. (editors), Temporal Databases: Theory, Design, and Implementation, Chapter 11, pages 248–270. Benjamin/Cummings, 1993.

[SSJ94] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 282–292, February 1994.

[Sto87] M. Stonebraker. The Design of the POSTGRES Storage System. In *Proceedings of the International Conference on Very Large Databases*, pages 289–300, Brighton, England, September 1987.