VLDB Journal (2003) / Digital Object Identifier (DOI) 10.1007/s00778-003-0111-3

The VLDB Journal, Online First, 2003.
URL: http://springerlink.metapress.com/link.asp?id=8geyeuu869m2twbp
The original publication is available at springerlink.com


# Join operations in temporal databases

**Dengfeng Gao**[1], **Christian S. Jensen**[2], **Richard T. Snodgrass**[1], **Michael D. Soo**[3]

[1] Computer Science Department, P.O. Box 210077, University of Arizona, Tucson, AZ 85721-0077, USA
e-mail: {dgao,rts}@cs.arizona.edu
[2] Department of Computer Science, Aalborg University, Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark
e-mail: csj@cs.auc.dk
[3] Amazon.com, Seattle; e-mail: soo@amazon.com

Edited by T. Sellis. Received: October 17, 2002 / Accepted: July 26, 2003
Published online: October 28, 2003 – © Springer-Verlag 2003

**Abstract.** Joins are arguably the most important relational operators. Poor implementations are tantamount to computing the Cartesian product of the input relations. In a temporal database, the problem is more acute for two reasons. First, conventional techniques are designed for the evaluation of joins with equality predicates rather than the inequality predicates prevalent in valid-time queries. Second, the presence of temporally varying data dramatically increases the size of a database. These factors indicate that specialized techniques are needed to efficiently evaluate temporal joins.

We address this need for efficient join evaluation in temporal databases. Our purpose is twofold. We first survey all previously proposed temporal join operators. While many temporal join operators have been defined in previous work, this work has been done largely in isolation from competing proposals, with little, if any, comparison of the various operators. We then address evaluation algorithms, comparing the applicability of various algorithms to the temporal join operators and describing a performance study involving algorithms for one important operator, the temporal equijoin. Our focus, with respect to implementation, is on non-index-based join algorithms. Such algorithms do not rely on auxiliary access paths but may exploit sort orderings to achieve efficiency.

**Keywords:** Attribute skew – Interval join – Partition join – Sort-merge join – Temporal Cartesian product – Temporal join – Timestamp skew

## 1 Introduction

Time is an attribute of all real-world phenomena. Consequently, efforts to incorporate the temporal domain into database management systems (DBMSs) have been ongoing for more than a decade [39,55]. The potential benefits of this research include enhanced data modeling capabilities and more conveniently expressed and efficiently processed queries over time.

Whereas most work in temporal databases has concentrated on conceptual issues such as data modeling and query languages, recent attention has been on implementation-related issues, most notably indexing and query processing strategies. In this paper, we consider an important subproblem of temporal query processing, the evaluation ad hoc temporal join operations, i.e., join operations for which indexing or secondary access paths are not available or appropriate. Temporal indexing, which has been a prolific research area in its own right [44], and query evaluation algorithms that exploit such temporal indexes are beyond the scope of this paper.

Joins are arguably the most important relational operators. This is so because efficient join processing is essential for the overall efficiency of a query processor. Joins occur frequently due to database normalization and are potentially expensive to compute [35]. Poor implementations are tantamount to computing the Cartesian product of the input relations. In a temporal database, the problem is more acute. Conventional techniques are aimed at the optimization of joins with equality predicates, rather than the inequality predicates prevalent in temporal queries [27]. Moreover, the introduction of a time dimension may significantly increase the size of the database. These factors indicate that new techniques are required to efficiently evaluate joins over temporal relations.

This paper aims to present a comprehensive and systematic study of join operations in temporal databases, including both semantics and implementation. Many temporal join operators have been proposed in previous research, but little comparison has been performed with respect to the semantics of these operators. Similarly, many evaluation algorithms supporting these operators have been proposed, but little analysis has appeared with respect to their relative performance, especially in terms of empirical study.

The main contributions of this paper are the following:

- To provide a systematic classification of temporal join operators as natural extensions of conventional join operators.
- To provide a systematic classification of temporal join evaluation algorithms as extensions of common relational query evaluation paradigms.
- To empirically quantify the performance of the temporal join algorithms for one important, frequently occurring, and potentially expensive temporal operator.

Our intention is for DBMS vendors to use the contributions of this paper as part of a migration path toward incorporating temporal support into their products. Specifically, we show that nearly all temporal query evaluation work to date has extended well-accepted conventional operators and evaluation algorithms. In many cases, these operators and techniques can be implemented with small changes to an existing code base and with acceptable, though perhaps not optimal, performance.

Research has identified two orthogonal dimensions of time in databases – *valid time*, modeling changes in the real world, and *transaction time*, modeling the update activity of the database [23,51]. A database may support none, one, or both of the given time dimensions. In this paper, we consider only single-dimension temporal databases, so-called *valid-time* and *transaction-time databases*. Databases supporting both time dimensions, so-called *bitemporal databases*, are beyond the scope of this paper, though many of the described techniques extend readily to bitemporal databases. We will use the terms *snapshot*, *relational*, or *conventional database* to refer to databases that provide no integrated support for time.

The remainder of the paper is organized as follows. We propose a taxonomy of temporal join operators in Sect. 2. This taxonomy extends well-established relational operators to the temporal context and classifies all previously defined temporal operators. In Sect. 3, we develop a corresponding taxonomy of temporal join evaluation algorithms, all of which are non-index-based algorithms. The next section focuses on engineering the algorithms. It turns out that getting the details right is essential for good performance. In Sect. 5, we empirically investigate the performance of the evaluation algorithms with respect to one particular, and important, valid-time join operator. The algorithms are tested under a variety of resource constraints and database parameters. Finally, conclusions and directions for future work are offered in Sect. 6.

## 2 Temporal join operators

In the past, temporal join operators were defined in different temporal data models; at times the essentially same operators were even given different names when defined in different models. Further, the existing join algorithms have also been constructed within the contexts of different data models. This section enables the comparison of join definitions and implementations across data models. We thus proceed to propose a taxonomy of temporal joins and then use this taxonomy to classify all previously defined temporal joins.

We take as our point of departure the core set of conventional relational joins that have long been accepted as "standard" [35]: Cartesian product (whose "join predicate" is the constant expression TRUE), theta join, equijoin, natural join, left and right outerjoin, and full outerjoin. For each of these, we define a temporal counterpart that is a natural, temporal generalization of it. This generalization hinges on the notion of snapshot equivalence [26], which states that two temporal relations are equivalent if they consist of the same sequence of time-indexed snapshots. We note that some other join operators do exist, including semijoin, antisemijoin, and difference. Their temporal counterparts have been explored elsewhere [11] and are not considered here.

Having defined this set of temporal joins, we show how all previously defined operators are related to this taxonomy of temporal joins. The previous operators considered include `Cartesian product`, $\Theta$-`JOIN`, `EQUIJOIN`, `NATURAL JOIN`, `TIME JOIN` [6,7], `TE JOIN`, `TE OUTERJOIN`, and `EVENT JOIN` [20,46,47,52] and those based on Allen's [1] interval relations ([27,28,36]). We show that many of these operators incorporate less restrictive predicates or use specialized attribute semantics and thus are variants of one of the taxonomic joins.

### 2.1 Temporal join definitions

To be specific, we base the definitions on a single data model. We choose the model that is used most widely in temporal data management implementations, namely, the one that timestamps each tuple with an interval. We assume that the timeline is partitioned into minimal-duration intervals, termed *chronons* [12], and we denote intervals by inclusive starting and ending chronons.

We define two temporal relational schemas, $R$ and $S$, as follows.

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

The $A_i$, $1 \leq i \leq n$ and $B_i$, $1 \leq i \leq m$ are the *explicit attributes* found in corresponding snapshot schemas, and $\mathrm{T}_s$ and $\mathrm{T}_e$ are the timestamp start and end attributes, recording when the information recorded by the explicit attributes holds (or held or will hold) true. We will use T as shorthand for the interval $[\mathrm{T}_s, \mathrm{T}_e]$ and $A$ and $B$ as shorthand for $\{A_1, \ldots, A_n\}$ and $\{B_1, \ldots, B_n\}$, respectively. Also, we define $r$ and $s$ to be instances of $R$ and $S$, respectively.

*Example 1* Consider the following two temporal relations. The relations show the canonical example of employees, the departments they work for, and the managers who supervise those departments.

Employee

| EmpName | Dept | T |
|---|---|---|
| Ron | Ship | [1,5] |
| George | Ship | [5,9] |
| Ron | Mail | [6,10] |

Manages

| Dept | MgrName | T |
|---|---|---|
| Load | Ed | [3,8] |
| Ship | Jim | [7,15] |

Tuples in the relations represent facts about the modeled reality. For example, the first tuple in the Employee relation represents the fact that Ron worked for the Shipping department from time 1 to time 5, inclusive. Notice that none of the attributes, including the timestamp attributes T, are set-valued – the relation schemas are in 1NF. □

### 2.2 Cartesian product

The temporal Cartesian product is a conventional Cartesian product with a predicate on the timestamp attributes. To define it, we need two auxiliary definitions.

First, $intersect(U, V)$, where $U$ and $V$ are intervals, returns TRUE if there exists a chronon $t$ such that

$t \in U \ \wedge \ t \in V$. Second, $overlap(U, V)$ returns the maximum interval contained in its two argument intervals. If no nonempty intervals exist, the function returns $\emptyset$. To state this more precisely, let *first* and *last* return the smallest and largest of two argument chronons, respectively. Also, let $U_s$ and $U_e$ denote, respectively, the starting and ending chronons of $U$, and similarly for $V$.

$$overlap(U,V) = \begin{cases} [last(U_s, V_s), first(U_e, V_e)] \\ \qquad \text{if } last(U_s, V_s) \leq first(U_e, V_e) \\ \emptyset \qquad \text{otherwise} \end{cases}$$

**Definition 1** The temporal Cartesian product, $r \times^{\mathrm{T}} s$, of two temporal relations $r$ and $s$ is defined as follows.

$$\begin{aligned} r \times^{\mathrm{T}} s = \{z^{(n+m+2)} \mid \exists x \in r \, \exists y \in s \, ( \\ z[A] = x[A] \wedge z[B] = y[B] \wedge \\ z[\mathrm{T}] = overlap(x[\mathrm{T}], y[\mathrm{T}]) \wedge z[\mathrm{T}] \neq \emptyset)\} \end{aligned}$$

The second line of the definition sets the explicit attribute values of the result tuple $z$ to the concatenation of the explicit attribute values of $x$ and $y$. The third line computes the timestamp of $z$ and ensures that it is nonempty. □

*Example 2* Consider the query "Show the names of employees and managers where the employee worked for the company while the manager managed some department in the company." This can be satisfied using the temporal Cartesian product.

Employee $\times^{\mathrm{T}}$ Manager

| EmpName | Dept | Dept | MgrName | T |
|---|---|---|---|---|
| Ron | Ship | Load | Ed | [3,5] |
| George | Ship | Load | Ed | [5,8] |
| George | Ship | Ship | Jim | [7,9] |
| Ron | Mail | Load | Ed | [6,8] |
| Ron | Mail | Ship | Jim | [7,10] |

□

The *overlap* function is necessary and sufficient to ensure *snapshot reducibility*, as will be discussed in detail in Sect. 2.7. Basically, we want the temporal Cartesian product to act as though it is a conventional Cartesian product applied independently at each point in time. When operating on interval-stamped data, this semantics corresponds to an intersection: the result will be valid during those times when contributing tuples from *both* input relations are valid.

The temporal Cartesian product was first defined by Segev and Gunadhi [20,47]. This operator was termed the `time join`, and the abbreviation `T-join` was used. Clifford and Croker [7] defined a Cartesian product operator that is a combination of the temporal Cartesian product and the temporal outerjoin, to be defined shortly. Interval join is a building block of the (spatial) rectangle join [2]. The interval join is a one-dimensional spatial join that can thus be used to implement the temporal Cartesian product.

### *2.3 Theta join*

Like the conventional theta join, the temporal theta join supports an unrestricted predicate $P$ on the explicit attributes of its input arguments. The temporal theta join, $r \bowtie^{\mathrm{T}}_P s$, of two relations $r$ and $s$ selects those tuples from $r \times^{\mathrm{T}} s$ that satisfy predicate $P(r[A], s[B])$. Let $\sigma$ denote the standard selection operator.

**Definition 2** The temporal theta join, $r \bowtie^{\mathrm{T}}_P s$, of two temporal relations $r$ and $s$ is defined as follows.

$r \bowtie^{\mathrm{T}}_P s = \sigma_{P(r[A],s[B])}(r \times^{\mathrm{T}} s)$ □

A form of this operator, the $\Theta$-`JOIN`, was proposed by Clifford and Croker [6]. This operator was later extended to allow computations more general than overlap on the timestamps of result tuples [53].

### *2.4 Equijoin*

Like snapshot equijoin, the temporal equijoin operator enforces equality matching among specified subsets of the explicit attributes of the input relations.

**Definition 3** The temporal equijoin on two temporal relations $r$ and $s$ on attributes $A' \subseteq A$ and $B' \subseteq B$ is defined as the theta join with predicate $P \equiv r[A'] = s[B']$.

$r \bowtie^{\mathrm{T}}_{r[A']=s[B']} s$ □

Like the temporal theta join, the temporal equijoin was first defined by Clifford and Croker [6]. A specialized operator, the `TE-join`, was developed independently by Segev and Gunadhi [47]. The `TE-join` required the explicit join attribute to be a surrogate attribute of both input relations. Essentially, a surrogate attribute would be a key attribute of a corresponding nontemporal schema. In a temporal context, a surrogate attribute value represents a time-invariant object identifier. If we augment schemas $R$ and $S$ with surrogate attributes *ID*, then the `TE-join` can be expressed using the temporal equijoin as follows.

$r$ `TE-join` $s \equiv r \bowtie^{\mathrm{T}}_{r[ID]=s[ID]} s$

The temporal equijoin was also generalized by Zhang et al. to yield the *generalized TE-join*, termed the *GTE-join*, which specifies that the joined tuples must have their keys in a specified range while their intervals should intersect a specified interval [56]. The objective was to focus on tuples within interesting rectangles in the key-time space.

### *2.5 Natural join*

The temporal natural join and the temporal equijoin bear the same relationship to one another as their snapshot counterparts. That is, the temporal natural join is simply a temporal equijoin on identically named explicit attributes followed by a subsequent projection operation.

To define this join, we augment our relation schemas with explicit join attributes, $C_i$, $1 \leq i \leq k$, which we abbreviate by $C$.

$$R = (A_1, \ldots, A_n, C_1, \ldots, C_k, \mathrm{T}_s, \mathrm{T}_e)$$
$$S = (B_1, \ldots, B_m, C_1, \ldots, C_k, \mathrm{T}_s, \mathrm{T}_e)$$

**Definition 4** The temporal natural join of $r$ and $s$, $r \bowtie^{\mathrm{T}} s$, is defined as follows.

$$r \bowtie^{\mathrm{T}} s = \{z^{(n+m+k+2)} \mid \exists x \in r\, \exists y \in s(x[C] = y[C] \wedge z[A] = x[A] \wedge z[B] = x[B] \wedge z[C] = y[C] \wedge z[\mathrm{T}] = overlap(x[\mathrm{T}], y[\mathrm{T}]) \wedge z[\mathrm{T}] \neq \emptyset)\}$$

The first two lines ensure that tuples $x$ and $y$ agree on the values of the join attributes $C$ and set the explicit attributes of the result tuple $z$ to the concatenation of the nonjoin attributes $A$ and $B$ and a single copy of the join attributes, $C$. The third line computes the timestamp of $z$ as the overlap of the timestamps of $x$ and $y$ and ensures that $x[\mathrm{T}]$ and $y[\mathrm{T}]$ actually overlap. □

This operator was first defined by Clifford and Croker [6], who named it the `natural time join`. We showed in earlier work that the temporal natural join plays the same important role in reconstructing normalized temporal relations as the snapshot natural join for normalized snapshot relations [25]. Most previous work in temporal join evaluation has addressed, either implicitly or explicitly, the implementation of the temporal natural join or the closely related temporal equijoin.

### *2.6 Outerjoins and outer Cartesian products*

Like the snapshot outerjoin, temporal outerjoins and Cartesian products retain *dangling tuples*, i.e., tuples that do not participate in the join. However, in a temporal database, a tuple may dangle over a portion of its time interval and be covered over others; this situation must be accounted for in a temporal outerjoin or Cartesian product.

We may define the temporal outerjoin as the union of two subjoins, like the snapshot outerjoin. The two subjoins are the temporal left outerjoin and the temporal right outerjoin. As the left and right outerjoins are symmetric, we define only the left outerjoin.

We need two auxiliary functions. The *coalesce* function collapses value-equivalent tuples – tuples with mutually equal nontimestamp attribute values [23] – in a temporal relation into a single tuple with the same nontimestamp attribute values and a timestamp that is the finite union of intervals that precisely contains the chronons in the timestamps of the value-equivalent tuples. (A finite union of time intervals is termed a *temporal element* [15], which we represent in this paper as a set of chronons.) The definition of *coalesce* uses the function *chronons* that returns the set of chronons contained in the argument interval.

$$coalesce(r) = \{z^{(n+1)} \mid \exists x \in r(z[A] = x[A] \Rightarrow chronons(x[\mathrm{T}]) \subseteq z[\mathrm{T}] \wedge \forall x' \in r(x[A] = x'[A] \Rightarrow (chronons(x'[\mathrm{T}]) \subseteq z[\mathrm{T}]))) \wedge \forall t \in z[\mathrm{T}]\, \exists x'' \in r(z[A] = x''[A] \wedge t \in chronons(x''[\mathrm{T}]))\}$$

The second and third lines of the definition coalesce all value-equivalent tuples in relation $r$. The last line ensures that no spurious chronons are generated.

We now define a function *expand* that returns the set of maximal intervals contained in an argument temporal element, $T$.

$$expand(T) = \{[t_s, t_e] \mid t_s \in T \wedge t_e \in T \wedge \forall t \in chronons([t_s, t_e])(t \in T) \wedge \neg\exists t'_s \in T\, (t'_s < t_s \wedge \forall t\, (t'_s < t < t_s \Rightarrow t \in T)) \wedge \neg\exists t'_e \in T\, (t'_e > t_e \wedge \forall t\, (t_e < t < t'_e \Rightarrow t \in T))\}$$

The second line ensures that a member of the result is an interval contained in $T$. The last two lines ensure that the interval is indeed maximal.

We are now ready to define the temporal left outerjoin. Let $R$ and $S$ be defined as for the temporal equijoin. We use $A' \subseteq A$ and $B' \subseteq B$ as the explicit join attributes.

**Definition 5** The temporal left outerjoin, $r \mathbb{X}^{\mathrm{T}}_{r[A']=s[B']} s$, of two temporal relations $r$ and $s$ is defined as follows.

$$r \mathbb{X}^{\mathrm{T}}_{r[A']=s[B']} s = \{z^{(n+m+2)} \mid \exists x \in coalesce(r)\, \exists y \in coalesce(s)\ (x[A'] = y[B'] \wedge z[A] = x[A] \wedge z[\mathrm{T}] \neq \emptyset \wedge ((z[B] = y[B] \wedge z[\mathrm{T}] \in expand(x[\mathrm{T}] \cap y[\mathrm{T}])) \vee (z[B] = \text{null} \wedge z[\mathrm{T}] \in expand(x[\mathrm{T}] - y[\mathrm{T}])))) \vee \exists x \in coalesce(r)\, \forall y \in coalesce(s)\ (x[A'] \neq y[B'] \Rightarrow z[A] = x[A] \wedge z[B] = \text{null} \wedge z[\mathrm{T}] \in expand(x[T]) \wedge z[\mathrm{T}] \neq \emptyset)\}$$

The first five lines of the definition handle the case where, for a tuple $x$ deriving from the left argument, a tuple $y$ with matching explicit join attribute values is found. For those time intervals of $x$ that are not shared with $y$, we generate tuples with null values in the attributes of $y$. The final three lines of the definition handle the case where no matching tuple $y$ is found. Tuples with null values in the attributes of $y$ are generated. □

The temporal outerjoin may be defined as simply the union of the temporal left and the temporal right outerjoins (the union operator eliminates the duplicate equijoin tuples). Similarly, a temporal outer Cartesian product is a temporal outerjoin without the equijoin condition ($A' = B' = \emptyset$).

Gunadhi and Segev were the first researchers to investigate outerjoins over time. They defined a specialized version of the temporal outerjoin called the `EVENT JOIN` [47]. This operator, of which the temporal left and right outerjoins were components, used a surrogate attribute as its explicit join attribute. This definition was later extended to allow any attributes to serve as the explicit join attributes [53]. A specialized version of the left and right outerjoins called the `TE-outerjoin` was also defined. The `TE-outerjoin` incorporated the `TE-join`, i.e., temporal equijoin, as a component.

Clifford and Croker [7] defined a temporal outer Cartesian product, which they termed simply Cartesian product.

### *2.7 Reducibility*

We proceed to show how the temporal operators reduce to snapshot operators. Reducibility guarantees that the semantics of the snapshot operator is preserved in its more complex temporal counterpart.

For example, the semantics of the temporal natural join reduces to the semantics of the snapshot natural join in that the result of first joining two temporal relations and then transforming the result to a snapshot relation yields a result that is the same as that obtained by first transforming the arguments to snapshot relations and then joining the snapshot relations. This commutativity diagram is shown in Fig. 1 and stated formally in the first equality of the following theorem.

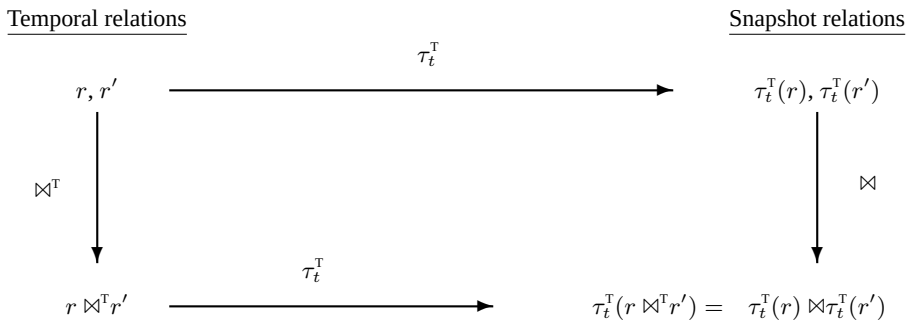


**Fig. 1.** Reducibility of temporal natural join to snapshot natural join

The timeslice operation $\tau^{\mathrm{T}}$ takes a temporal relation $r$ as argument and a chronon $t$ as parameter. It returns the corresponding snapshot relation, i.e., with the schema of $r$ but without the timestamp attributes, that contains (the nontime stamp portion of) all tuples $x$ from $r$ for which $t$ belongs to $x[\mathrm{T}]$. It follows from the theorem below that the temporal joins defined here reduce to their snapshot counterparts.

**Theorem 1** *Let $t$ denote a chronon and let $r$ and $s$ be relation instances of the proper types for the operators they are applied to. Then the following hold for all $t$.*

$$\tau_t^{\mathrm{T}}(r \bowtie^{\mathrm{T}} s) = \tau_t^{\mathrm{T}}(r) \bowtie \tau_t^{\mathrm{T}}(s)$$
$$\tau_t^{\mathrm{T}}(r \times^{\mathrm{T}} s) = \tau_t^{\mathrm{T}}(r) \times \tau_t^{\mathrm{T}}(s)$$
$$\tau_t^{\mathrm{T}}(r \bowtie_P^{\mathrm{T}} s) = \tau_t^{\mathrm{T}}(r) \bowtie_P \tau_t^{\mathrm{T}}(s)$$
$$\tau_t^{\mathrm{T}}(r \rtimes^{\mathrm{T}} s) = \tau_t^{\mathrm{T}}(r) \rtimes \tau_t^{\mathrm{T}}(s)$$
$$\tau_t^{\mathrm{T}}(r \ltimes^{\mathrm{T}} s) = \tau_t^{\mathrm{T}}(r) \ltimes \tau_t^{\mathrm{T}}(s)$$

PROOF: An equivalence is shown by proving its two inclusions separately. The nontimestamp attributes of $r$ and $s$ are $AC$ and $BC$, respectively, where $A$, $B$, and $C$ are sets of attributes and $C$ denotes the join attribute(s) (cf. the definition of temporal natural join). We prove one inclusion of the first equivalence, that is, $\tau_t^{\mathrm{T}}(r \bowtie^{\mathrm{T}} s) \subseteq \tau_t^{\mathrm{T}}(r) \bowtie \tau_t^{\mathrm{T}}(s)$. The remaining proofs are similar in style.

Let $x'' \in \tau_t^{\mathrm{T}}(r \bowtie s)$ (the left-hand side of the equivalence to be proved). Then there is a tuple $x' \in r \bowtie^{\mathrm{T}} s$ such that $x'[ABC] = x''$ and $t \in x'[\mathrm{T}]$. By the definition of $\bowtie^{\mathrm{T}}$, there exist tuples $x_1 \in r$ and $x_2 \in s$ such that $x_1[C] = x_2[C] = x'[C]$, $x_1[A] = x'[A]$, $x_2[B] = x'[B]$, $x'[T] = overlap(x_1[T], x_2[T])$. By the definition of $\tau_t^{\mathrm{T}}$, there exist a tuple $x_1' \in \tau_t^{\mathrm{T}}(r)$ such that $x_1' = x_1[AC] = x'[AC]$ and a tuple $x_2' \in \tau_t^{\mathrm{T}}(s)$ such that $x_2' = x_2[BC] = x'[BC]$. Then there exists $x_{12}'' \in \tau_t^{\mathrm{T}}(r) \bowtie \tau_t^{\mathrm{T}}(s)$ (the right-hand side of the equivalence) such that $x_{12}''[AC] = x_1'$ and $x_{12}''[B] = x_2'[B]$. By construction, $x_{12}'' = x''$. This proves the $\subseteq$ inclusion. □

### *2.8 Summary*

We have defined a taxonomy for temporal join operators. The taxonomy was constructed as a natural extension of corresponding snapshot database operators. We also briefly described how previously defined temporal operators are accommodated in the taxonomy.

Table 1 summarizes how previous work is represented in the taxonomy. For each operator defined in previous work, the table lists the defining publication, researchers, the corresponding taxonomy operator, and any restrictions assumed by the original operators. In early work, Clifford [8] indicated that an `INTERSECTION JOIN` should be defined that represents the categorized nonouter joins and Cartesian products, and he proposed that a `UNION JOIN` be defined for the outer variants.

## 3 Evaluation algorithms

In the previous section, we described the semantics of all previously proposed temporal join operators. We now turn our attention to implementation algorithms for these operators. As before, our purpose is to enumerate the space of algorithms applicable to the temporal join operators, thereby providing a consistent framework within which existing temporal join evaluation algorithms can be placed.

Our approach is to extend well-understood paradigms from conventional query evaluation to temporal databases. Algorithms for temporal join evaluation are necessarily more complex than their snapshot counterparts. Whereas snapshot evaluation algorithms match input tuples based on their explicit join attributes, temporal join evaluation algorithms typically must additionally ensure that temporal restrictions are met. Furthermore, this problem is exacerbated in two ways. Timestamps are typically complex data types, e.g., intervals requiring inequality predicates, which conventional query processors are not optimized to handle. Also, a temporal database is usually larger than a corresponding snapshot database due to the versioning of tuples.

We consider non-index-based algorithms. Index-based algorithms use an auxiliary access path, i.e., a data structure that identifies tuples or their locations using a join attribute value. Non-index-based algorithms do not employ auxiliary access paths. While some attention has been focused on index-based temporal join algorithms, the large number of temporal indexes that have been proposed in the literature [44] precludes a thorough investigation in this paper.

We first provide a taxonomy of temporal join algorithms. This taxonomy, like the operator taxonomy of Table 1, is based on well-established relational concepts. Sections 3.2 and 3.3 describe the algorithms in the taxonomy and place existing work within the given framework. Finally, conclusions are offered in Sect. 3.4.

### *3.1 Evaluation taxonomy*

All binary relational query evaluation algorithms, including those computing conventional joins, are derived from four

**Table 1.** Temporal join operators

| Operator | Initial citation | Taxonomy operator | Restrictions |
|---|---|---|---|
| Cartesian product | [7] | Outer Cartesian product | None |
| EQUIJOIN | [6] | Equijoin | None |
| GTE-join | [56] | Equijoin | 2, 3 |
| INTERVAL JOIN | [2] | Cartesian product | None |
| NATURAL JOIN | [6] | Natural join | None |
| TIME JOIN | [6] | Cartesian product | 1 |
| T-join | [20] | Cartesian product | None |
| TE-JOIN | [47] | Equijoin | 2 |
| TE-OUTERJOIN | [47] | Left outerjoin | 2 |
| EVENT JOIN | [47] | Outerjoin | 2 |
| $\Theta$-JOIN | [6] | Theta join | None |
| Valid-time theta join | [53] | Theta join | None |
| Valid-time left join | [53] | Left outerjoin | None |

*Restrictions:*
1 = restricts also the valid time of the result tuples
2 = matching only on surrogate attributes
3 = includes also intersection predicates with an argument surrogate range and a time range

basic paradigms: nested-loop, partitioning, sort-merge, and index-based [18].

Partition-based join evaluation divides the input tuples into buckets using the join attributes of the input relations as key values. Corresponding buckets of the input relations contain all tuples that could possibly match with one another, and the buckets are constructed to best utilize the available main memory buffer space. The result is produced by performing an in-memory join of each pair of corresponding buckets from the input relations.

Sort-merge join evaluation also divides the input relation but uses physical memory loads as the units of division. The memory loads are sorted, producing sorted runs, and written to disk. The result is produced by merging the sorted runs, where qualifying tuples are matched and output tuples generated.

Index-based join evaluation utilizes indexes defined on the join attributes of the input relations to locate joining tuples efficiently. The index could be preexisting or built on the fly. Elmasri et al. presented a temporal join algorithm that utilizes a two-level time index, which used a $B^+$-tree to index the explicit attribute in the upper level, with the leaves referencing other $B^+$-trees indexing time points [13]. Son and Elmasri revised the time index to require less space and used this modified index to determine the partitioning intervals in a partition-based timestamp algorithm [52]. Bercken and Seeger proposed several temporal join algorithms based on a multiversion $B^+$-tree (MVBT) [4]. Later Zhang et al. described several algorithms based on $B^+$-trees, $R^*$-trees [3], and the MVBT for the related GTE-join. This operation requires that joined tuples have key values that belong to a specified range and have time intervals that intersect a specified interval [56]. The MVBT assumes that updates arrive in increasing time order, which is not the case for valid-time data. We focus on non-index-based join algorithms that apply to both valid-time and transaction-time relations, and we do not discuss these index-based joins further.

We adapt the basic non-index-based algorithms (nested-loop, partitioning, and sort-merge) to support temporal joins. To enumerate the space of temporal join algorithms, we exploit the duality of partitioning and sort-merge [19]. In particular, the division step of partitioning, where tuples are separated based on key values, is analogous to the merging step of sort-merge, where tuples are matched based on key values. In the following, we consider the characteristics of sort-merge algorithms and apply duality to derive corresponding characteristics of partition-based algorithms.

For a conventional relation, sort-based join algorithms order the input relation on the input relations' explicit join attributes. For a temporal relation, which includes timestamp attributes in addition to explicit attributes, there are four possibilities for ordering the relation. First, the relation can be sorted by the explicit attributes exclusively. Second, the relation can be ordered by time, using either the starting or ending timestamp [29,46]. The choice of starting or ending timestamp dictates an ascending or descending sort order, respectively. Third, the relation can be ordered primarily by the explicit attributes and secondarily by time [36]. Finally, the relation can be ordered primarily by time and secondarily by the explicit attributes.

By duality, the division step of partition-based algorithms can partition using any of these options [29,46]. Hence four choices exist for the dual steps of merging in sort-merge or partitioning in partition-based methods.

We use this distinction to categorize the different approaches to temporal join evaluation. The first approach above, using the explicit attributes as the primary matching attributes, we term *explicit algorithms*. Similarly, we term the second approach *timestamp algorithms*. We retain the generic term *temporal algorithm* to mean any algorithm to evaluate a temporal operator.

Finally, it has been recognized that the choice of buffer allocation strategy, GRACE or hybrid [9], is independent of whether a sort-based or partition-based approach is used [18]. Hybrid policies retain most of the last run of the outer relation in main memory and so minimize the flushing of intermediate buffers to disk, thereby potentially decreasing the I/O cost.

Figure 2 lists the choices of sort-merge vs. partitioning, the possible sorting/partitioning attributes, and the possible

$$\left\{\begin{matrix}\text{Sort-merge}\\ \text{Partitioning}\end{matrix}\right\} \times \left\{\begin{matrix}\text{Explicit}\\ \text{Timestamp}\\ \text{Explicit/timestamp}\\ \text{Timestamp/explicit}\end{matrix}\right\} \times \left\{\begin{matrix}\text{GRACE}\\ \text{Hybrid}\end{matrix}\right\}$$

**Fig. 2.** Space of possible evaluation algorithms

buffer allocation strategies. Combining all possibilities yields 16 possible evaluation algorithms. Including the basic nested-loop algorithm and GRACE and hybrid variants of the sort-based interval join mentioned in Sect. 2.2 results in a total of 19 possible algorithms. The 19 algorithms are named and described in Table 2.

We noted previously that time intervals lack a natural order. From this point of view spatial join is similar because there is no natural order preserving spatial closeness. Previous work on spatial join may be categorized into three approaches. Early work [37,38] used a transformation approach based on space-filling curves, performing a sort-merge join along the curve to solve the join problem. Most of the work falls in the index-based approaches, utilizing spatial index structures such as the R-tree [21], $R^+$-tree [48], $R^*$-tree [3], Quad-tree [45], or seeded tree [31]. While some algorithms use preexisting indexes, others build the indexes on the fly.

In recent years, some work has focused on non-index-based spatial join approaches. Two partition-based spatial join algorithms have been proposed. One of them [32] partitions the input relations into overlapping buckets and uses an indexed nested-loop join to perform the join within each bucket. The other [40] partitions the input relations into disjoint partitions and uses a computational-geometry-based plane-sweep algorithm that can be thought of as the spatial equivalent of the sort-merge algorithm. Arge et al. [2] introduced a highly optimized implementation of the sweeping-based algorithm that first sorts the data along the vertical axis and then partitions the input into a number of vertical strips. Data in each strip can then be joined by an internal plane-sweep algorithm. All the above non-index-based spatial join algorithms use a sort- or partition-based approach or combine these two approaches in one algorithm, which is the approach we adopt in some of our temporal join algorithms (Sect. 4.3.2).

In the next two sections, we examine the space of explicit algorithms and timestamp algorithms, respectively, and classify existing approaches using the taxonomy developed in this section. We will see that most previous work in temporal join evaluation has centered on timestamp algorithms. However, for expository purposes, we first examine those algorithms based on manipulation of the nontimestamp columns, which we term "explicit" algorithms.

### *3.2 Explicit algorithms*

Previous work has largely ignored the fact that conventional query evaluation algorithms can be easily modified to evaluate temporal joins. In this section, we show how the three paradigms of query evaluation can support temporal join evaluation. To make the discussion concrete, we develop an algorithm to evaluate the valid-time natural join, defined in Sect. 2, for each of the three paradigms. We begin with the simplest paradigm, nested-loop evaluation.

```
explicitNestedLoop(r, s):
    result ← ∅;
    for each block b_r ∈ r
        read(b_r);
        for each block b_s ∈ s
            read(b_s);
            for each tuple x ∈ b_r
                for each tuple y ∈ b_s
                    if x[C] = y[C] and
                        overlap(x[T], y[T]) ≠ ∅
                        z[A] ← x[A];
                        z[B] ← y[B];
                        z[C] ← x[C];
                        z[T] ← overlap(x[T], y[T]);
                        result ← result ∪ {z};
    return result;
```

**Fig. 3.** Algorithm `explicitNestedLoop`

#### 3.2.1 Nested-loop-based algorithms

Nested-loop join algorithms match tuples by exhaustively comparing pairs of tuples from the input relations. As an I/O optimization, blocks of the input relations are read into memory, with comparisons performed between all tuples in the input blocks. The size of the input blocks is constrained by the available main memory buffer space.

The algorithm operates as follows. One relation is designated the *outer relation*, the other the *inner relation* [35,18]. The outer relation is scanned once. For each block of the outer relation, the inner relation is scanned. When a block of the inner relation is read into memory, the tuples in that "inner block" are joined with the tuples in the "outer block."

The temporal nested-loop join is easily constructed from this basic algorithm. All that is required is that the timestamp predicate be evaluated at the same time as the predicate on the explicit attributes. Figure 3 shows the temporal algorithm. (In the figure, $r$ is the outer relation and $s$ is the inner relation. We assume their schemas are as defined in Sect. 2.)

While conceptually simple, nested-loop-based evaluation is often not competitive due to its quadratic cost. We now describe temporal variants of the sort-merge and partition-based algorithms, which usually exhibit better performance.

#### 3.2.2 Sort-merge-based algorithms

Sort-merge join algorithms consist of two phases. In the first phase, the input relations $r$ and $s$ are sorted by their join attributes. In the second phase, the result is produced by simultaneously scanning $r$ and $s$, merging tuples with identical values for their join attributes.

Complications arise if the join attributes are not key attributes of the input relations. In this case, multiple tuples in $r$ and in $s$ may have identical join attribute values. Hence a given $r$ tuple may join with many $s$ tuples, and vice versa. (This is termed *skew* [30].)

As before, we designate one relation as the outer relation and the other as the inner relation. When consecutive tuples in

**Table 2.** Algorithm taxonomy

| Algorithm | Name | Description |
|---|---|---|
| Explicit sort | ES | GRACE sort-merge by explicit attributes |
| Hybrid explicit sort | ES-H | Hybrid sort-merge by explicit attributes |
| Timestamp sort | TS | GRACE sort-merge by timestamps |
| Hybrid timestamp sort | TS-H | Hybrid sort-merge by timestamps |
| Explicit/timestamp sort | ETS | GRACE sort-merge by explicit attributes/time |
| Hybrid explicit/timestamp sort | ETS-H | Hybrid sort-merge by explicit attributes/time |
| Timestamp/explicit sort | TES | GRACE sort-merge by time/explicit attributes |
| Hybrid timestamp/explicit sort | TES-H | Hybrid sort-merge by time/explicit attributes |
| Interval join | TSI | GRACE sort-merge by timestamps |
| Hybrid interval join | TSI-H | Hybrid sort-merge by timestamps |
| Explicit partitioning | EP | GRACE partitioning by explicit attributes |
| Hybrid explicit partitioning | EP-H | Hybrid partitioning by explicit attributes |
| Timestamp partitioning | TP | Range partition by time |
| Hybrid timestamp partitioning | TP-H | Hybrid range partitioning by time |
| Explicit/timestamp partitioning | ETP | GRACE partitioning by explicit attributes/time |
| Hybrid explicit/timestamp partitioning | ETP-H | Hybrid partitioning by explicit attributes/time |
| Timestamp/explicit partitioning | TEP | GRACE partitioning by time/explicit attributes |
| Hybrid timestamp/explicit partitioning | TEP-H | Hybrid partitioning by time/explicit attributes |
| Nested-loop | NL | Exhaustive matching |

```
structure state
    integer current_block;
    integer current_tuple;
    integer first_block;
    integer first_tuple;
    block tuples;
```

**Fig. 4.** State structure for merge scanning

the outer relation have identical values for their explicit join attributes, i.e., their nontimestamp join attributes, the scan of the inner relation is "backed up" to ensure that all possible matches are found. Prior to showing the `explicitSortMerge` algorithm, we define a suite of algorithms that manage the scans of the input relations. For each scan, we maintain the state structure shown in Fig. 4. The fields *current_block* and *current_tuple* together indicate the current tuple in the scan by recording the number of the current block and the index of the current tuple within that block. The fields *first_block* and *first_tuple* are used to record the state at the beginning of a scan of the inner relation in order to back up the scan later if needed. Finally, *tuples* stores the block of the relation currently in memory. For convenience, we treat the block as an array of tuples.

The `initState` algorithm shown in Fig. 5 initializes the state of a scan. Essentially, counters are set to guarantee that the first block read and the first tuple scanned are the first block and first tuple within that block in the input relation. We assume that a `seek` operation is available that repositions the file pointer associated with a relation to a given block number.

The `advance` algorithm advances the scan of the argument relation and state to the next tuple in the sorted relation. If the current block has been exhausted, then the next block of the relation is read. Otherwise, the state is updated to mark the next tuple in the current block as the next tuple in the scan. The

```
initState(relation, state):
    state.current_block ← 1;
    state.current_tuple ← 0;
    state.first_block ←⊥;
    state.first_tuple ←⊥;
    seek(relation, state.current_block);
    state.tuples ← read_block(relation);

advance(relation, state):
    if (state.current_tuple = MAX_TUPLES)
        state.tuples ← read_block(relation);
        state.current_block ← state.current_block + 1;
        state.current_tuple ← 1;
    else
        state.current_tuple ← state.current_tuple + 1;

currentTuple(state):
    return state.tuples[state.current_tuple]

backUp(relation, state):
    if (state.current_block ≠ state.first_block)
        state.current_block ← state.first_block;
        seek(relation, state.current_block);
        state.tuples ← read_block(relation);
    state.current_tuple ← state.first_tuple;

markScanStart(state):
    state.first_block ← state.current_block;
    state.first_tuple ← state.current_tuple;
```

**Fig. 5.** Merge algorithms

```
explicitSortMerge(r, s, C):
    r' ← sort(r, C);
    s' ← sort(s, C);

    initState(r', outer_state); initState(s', inner_state);
    x'[C] ←⊥;
    result ← ∅;
    advance(s', inner_state);
    y ← currentTuple(inner_state);


    for i ← 1 to |r'|
        advance(r', outer_state);
        x ← currentTuple(outer_state);

        if x[C] = x'[C]
            backUp(s', inner_state);
            y ← currentTuple(s', inner_state);
        x'[C] ← x[C];

        while (x[C] > y[C])
            advance(s', inner_state);
            y ← currentTuple(inner_state);

        markScanStart(inner_state);

        while (x[C] = y[C])
            if overlap(x[T], y[T]) ≠ ∅)
                z[A] ← x[A]; z[B] ← y[B]; z[C] ← x[C];
                z[T] ← overlap(x[T], y[T]);
                result ← result ∪ {z};
            advance(s', inner_state);
            y ← currentTuple(inner_state);

    return result;
```

**Fig. 6.** `explicitSortMerge` algorithm

`current_tuple` algorithm merely returns the next tuple in the scan, as indicated by the scan state. Finally, the `backUp` and `markScanStart` algorithms manage the backing up of the inner relation scan. The `backUp` algorithm reverts the current block and tuple counters to their last values. These values are stored in the state at the beginning of a scan by the `markScanStart` algorithm.

We are now ready to exhibit the `explicitSortMerge` algorithm, shown in Fig. 6. The algorithm accepts three parameters, the input relations $r$ and $s$ and the join attributes $C$. We assume that the schemas of $r$ and $s$ are as given in Sect. 2. Tuples from the outer relation are scanned in order. For each outer tuple, if the tuple matches the previous outer tuple, the scan of the inner relation is backed up to the first matching inner tuple. The starting location of the scan is recorded in case backing up is needed by the next outer tuple, and the scan proceeds forward as normal. The complexity of the algorithm, as well as its performance degradation as compared with conventional sort-merge, is due largely to the bookkeeping required to back up the inner relation scan. We consider this performance hit in more detail in Sect. 4.2.2.

Segev and Gunadhi developed three algorithms based on explicit sorting, differing primarily by the code in the inner loop and by whether backup is necessary. Two of the algorithms, TEJ-1 and TEJ-2, support the temporal equijoin [46]; the remaining algorithm, EJ-1, evaluates the temporal outerjoin [46].

TEJ-1 is applicable if the equijoin condition is on the surrogate attributes of the input relations. The surrogate attributes are essentially key attributes of a corresponding snapshot schema. TEJ-1 assumes that the input relations are sorted primarily by their surrogate attributes and secondarily by their starting timestamps. The surrogate matching, sort-ordering, and 1TNF assumption described in Sect. 3.3.1 allows the result to be produced with a single scan of both input relations, with no backup.

The second equijoin algorithm, TEJ-2, is applicable when the equijoin condition involves any explicit attributes, surrogate or not. TEJ-2 assumes that the input relations are sorted primarily by their explicit join attribute(s) and secondarily by their starting timestamps. Note that since the join attribute can be a nonsurrogate attribute, tuples sharing the same join attribute value may overlap in valid time. Consequently, TEJ-2 requires the scan of the inner relation to be backed up in order to find all tuples with matching explicit attributes.

For the EVENT JOIN, Segev and Gunadhi described the sort-merge-based algorithm EJ-1. EJ-1 assumes that the input relations are sorted primarily by their surrogate attributes and secondarily by their starting timestamps. Like TEJ-1, the result is produced by a single scan of both input relations.

### 3.2.3 Partition-based algorithms

As in sort-merge-based algorithms, partition-based algorithms have two distinct phases. In the first phase, the input relations are partitioned based on their join attribute values. The partitioning is performed so that a given bucket produced from one input relation contains tuples that can only match with tuples contained in the corresponding bucket of the other input relation. Each produced bucket is also intended to fill the allotted main memory. Typically, a hash function is used as the partitioning agent. Both relations are filtered through the same hash function, producing two parallel sets of buckets. In the second phase, the join is computed by comparing tuples in corresponding buckets of the input relations. Partition-based algorithms have been shown to have superior performance when the relative sizes of the input relations differ [18].

A partitioning algorithm for the temporal natural join is shown in Fig. 7. The algorithm accepts as input two relations $r$ and $s$ and the names of the explicit join attributes $C$. We assume that the schemas of $r$ and $s$ are as given in Sect. 2.

As can be seen, the explicit partition-based join algorithm is conceptually very simple. One relation is designated the outer relation, the other the inner relation. After partitioning, each bucket of the outer relation is read in turn. For a given "outer bucket," each page of the corresponding "inner bucket" is read, and tuples in the buffers are joined.

The partitioning step in Fig. 7 is performed by the `partition` algorithm. This algorithm takes as its first argument an input relation. The resulting $n$ partitions are returned in the remaining parameters. Algorithm partition assumes that a hash function `hash` is available that accepts the join attribute values $x[C]$ as input and returns an integer, the index of the target bucket, as its result.

```
explicitPartitionJoin(r, s, C):
    result ← ∅;

    partition(r, r_1, ..., r_n);
    partition(s, s_1, ..., s_n);

    for i ← 1 to n
        outer_bucket ← read_partition(r_i);
        for each page p ∈ s_i
            p ← read_page(s_i);
            for each tuple x ∈ outer_bucket
                for each tuple y ∈ p
                    if (x[C] = y[C] and
                        overlap(x[T], y[T]) ≠ ∅)
                        z[A] ← x[A];
                        z[B] ← y[B];
                        z[C] ← x[C];
                        z[T] ← overlap(x[T], y[T]);
                        result ← result ∪ {z};
    return result;


partition(r, r_1, ..., r_n):
    for i ← 1 to p
        r_i ← ∅;

    for each block b ∈ r
        read_block(b);
        for each tuple x ∈ b
            i ← hash(x[C]);
            r_i ← r_i ∪ {x};
```

**Fig. 7.** Algorithms `explicitPartitionJoin` and `partition`

### *3.3 Timestamp algorithms*

In contrast to the algorithms of the previous section, timestamp algorithms perform their primary matching on the timestamps associated with tuples.

In this section, we enumerate, to the best of our knowledge, all existing timestamp-based evaluation algorithms for the temporal join operators described in Sect. 3. Many of these algorithms assume sort ordering of the input by either their starting or ending timestamps. While such assumptions are valid for many applications, they are not valid in the general case, as valid-time semantics allows correction and deletion of previously stored data. (Of course, in such cases one could resort *within* the join.) As before, all of the algorithms described here are derived from nested loop, sort-merge, or partitioning; we do not consider index-based temporal joins.

#### 3.3.1 Nested-loop-based timestamp algorithms

One timestamp nested-loop-based algorithm has been proposed for temporal join evaluation. Like the EJ-1 algorithm described in the previous section, Segev and Gunadhi developed their algorithm, EJ-2, for the EVENT JOIN [47, 20] (Table 1).

EJ-2 does not assume any ordering of the input relations. It does assume that the explicit join attribute is a distinguished surrogate attribute and that the input relations are in Temporal First Normal Form (1TNF). Essentially, 1TNF ensures that tuples within a single relation that have the same surrogate value may not overlap in time.

EJ-2 simultaneously produces the natural join and left outerjoin in an initial phase and then computes the right outerjoin in a subsequent phase.

For the first phase, the inner relation is scanned once from front to back for each outer relation tuple. For a given outer relation tuple, the scan of the inner relation is terminated when the inner relation is exhausted or the outer tuple's timestamp has been completely overlapped by matching inner tuples. The outer tuple's natural join is produced as the scan progresses. The outer tuple's left outerjoin is produced by tracking the subintervals of the outer tuple's timestamp that are not overlapped by any inner tuples. An output tuple is produced for each subinterval remaining at the end of the scan. Note that the main memory buffer space must be allocated to contain the nonoverlapped subintervals of the outer tuple.

In the second phase, the roles of the inner and outer relations are reversed. Now, since the natural join was produced during the first phase, only the right outerjoin needs to be computed. The right outerjoin tuples are produced in the same manner as above, with one small optimization. If it is known that a tuple of the (current) outer relation did not join with any tuples during the first phase, then no scanning of the inner relation is required and the corresponding outerjoin tuple is produced immediately.

Incidentally, Zurek proposed several algorithms for evaluating temporal *Cartesian product* on multiprocessors based on nested loops [57].

#### 3.3.2 Sort-merge-based timestamp algorithms

To date, four sets of researchers – Segev and Gunadhi, Leung and Muntz, Pfoser and Jensen, and Rana and Fotouhi – have developed timestamp sort-merge algorithms. Additionally, a one-dimensional spatial join algorithm proposed by Arge et al. can be used to implement a temporal Cartesian product.

Segev and Gunadhi modified the traditional merge-join algorithm to support the T-join and the temporal equijoin [47, 20]. We describe the algorithms for each of these operators in turn.

For the T-join, the relations are sorted in ascending order of starting timestamp. The result is produced by a single scan of the input relations.

For the temporal equijoin, two timestamp sorting algorithms, named TEJ-3 and TEJ-4, are presented. Both TEJ-3 and TEJ-4 assume that their input relations are sorted by starting timestamp only. TEJ-4 is applicable only if the equijoin condition is on the surrogate attribute. In addition to assuming that the input relations are sorted by their starting timestamps, TEJ-4 assumes that all tuples with the same surrogate value are linked, thereby allowing all tuples with the same surrogate to be retrieved when the first is found. The result is performed with a linear scan of both relations, with random access needed to traverse surrogate chains.

Like TEJ-2, TEJ-3 is applicable for temporal equijoins on both the surrogate and explicit attribute values. TEJ-3 assumes that the input relations are sorted in ascending order of

their starting timestamps, but no sort order is assumed on the explicit join attributes. Hence TEJ-3 requires that the inner relation scan be backed up should consecutive tuples in the outer relation have overlapping interval timestamps.

Leung and Muntz developed a series of algorithms based on the sort-merge algorithm to support temporal join predicates such as "contains" and "intersect" [1]. Although their algorithms do not explicitly support predicates on nontemporal attribute values, their techniques are easily modified to support more complex join operators such as the temporal equijoin. Like Segev and Gunadhi, this work describes evaluation algorithms appropriate for different sorting assumptions and access paths.

Leung and Muntz use a stream-processing approach. Abstractly, the input relations are considered as sequences of time-sorted tuples where only the tuples at the front of the streams may be read. The ordering of the tuples is a tradeoff with the amount of main memory needed to compute the join. For example, Leung and Muntz show how a contain join [1] can be computed if the input streams are sorted in ascending order of their starting timestamp. They summarize for various sort orders on the starting and ending timestamps what tuples must be retained in main memory during the join computation. A family of algorithms are developed assuming different orderings (ascending/descending) of the starting and ending timestamps.

Leung and Muntz also show how checkpoints, essentially the set of tuples valid during some chronon, can be used to evaluate temporal joins where the join predicate implies some overlap between the participating tuples. Here, the checkpoints actually contain tuple identifiers (TIDs) for the tuples valid during the specified chronon and the TIDs of the next tuples in the input streams. Suppose a checkpoint exists at time $t$. Using this checkpoint, the set of tuples participating in a join over a time interval containing $t$ can be computed by using the cached TIDs and "rolling forward" using the TIDs of the next tuples in the streams.

Rana and Fotouhi proposed several techniques to improve the performance of time-join algorithms in which they claimed they used a nested-loop approach [43]. Since they assumed the input relations were sorted by the start time and/or end time, those algorithms are more like the second phase of sort-merge-based timestamp algorithms. The algorithms are very similar to the sort-merge-based algorithms developed by Segev and Gunadhi.

Arge et al. described the interval join, a one-dimensional spatial join algorithm, which is a building block of a two-dimensional rectangle join [2]. Each interval is defined by a lower boundary and an upper boundary. The problem is to report all intersections between an interval in the outer relation and an interval in the inner relation. If the interval is a time interval instead of a spatial interval, this problem is equivalent to the temporal Cartesian product. They assumed the two input relations were first sorted by the algorithm into one list by their lower boundaries. The algorithm maintains two initially empty lists of tuples with "active" intervals, one for each input relation. When the sorted list is scanned, the current tuple is put into the active list of the relation it belongs to and joins only with the tuples in the active list of the other relation. Tuples becoming inactive during scanning are removed from the active list.

Most recently, Pfoser and Jensen [41] applied the sort-merge approach to the temporal theta join in a setting where each argument relation consists of a noncurrent and a current partition. Tuples in the former all have intervals that end before the current time, while all tuples of the latter have intervals that end at the current time. They assume that updates arrive in time order, so that tuples in noncurrent partitions are ordered by their interval end times and tuples in current partitions are ordered by their interval start times. A join then consists of three different kinds of subjoins. They develop two join algorithms for this setting and subsequently use these algorithms for incremental join computation.

As can be seen from the above discussion, a large number of timestamp-based sort-merge algorithms have been proposed, some for specific join operators. However, each of these proposals has been developed largely in isolation from other work, with little or no cross comparison. Furthermore, published performance figures have been derived mainly from analytical models rather than from empirical observations. An empirical comparison, as provided in Sect. 5, is needed to truly evaluate the different proposals.

#### 3.3.3 Partition-based timestamp algorithms

Partitioning a relation over explicit attributes is relatively straightforward if the partitioning attributes have discrete values. Partitioning over time is more difficult since our timestamps are intervals, i.e., range data, rather than discrete values. Previous timestamp partitioning algorithms therefore developed various means of *range partitioning* the time intervals associated with tuples.

In previous work, we described a valid-time join algorithm using partitioning [54]. This algorithm was presented in the context of evaluating the valid-time natural join, though it is easily adapted to other temporal joins. The range partitioning used by this algorithm mapped tuples to singular buckets and dynamically migrated the tuples to other buckets as needed during the join computation. This approach avoided data redundancy, and associated I/O overhead, at the expense of more complex buffer management.

Sitzmann and Stuchey extended this algorithm by using histograms to decide the partition boundary [49]. Their algorithm takes the number of long-lived tuples into consideration, which renders its performance insensitive to the number of long-lived tuples. However, it relies on a preexisting temporal histogram.

Lu et al. described another range-partitioning algorithm for computing temporal joins [33]. This algorithm is applicable to theta joins, where a result tuple is produced for each pair of input tuples with overlapping valid-time intervals. Their approach is to map intervals to a two-dimensional plane, which is then partitioned into regions. The join result is produced by computing the subjoins of pairs of partitions corresponding to adjacent regions in the plane. This method applies to a restricted temporal model where future time is not allowed. They utilize a spatial index to speed up the joining phase.

**Table 3.** Existing algorithms and taxonomy counterparts

| Algorithm | Defined by | Taxonomy | Assumptions |
|---|---|---|---|
| TEJ-1 | Segev and Gunadhi | Explicit/timestamp sort | Surrogate attribute and 1TNF |
| TEJ-2 | Segev and Gunadhi | Explicit/timestamp sort | None |
| EJ-2 | Segev and Gunadhi | Nested-loop | Surrogate attribute and 1TNF |
| EJ-1 | Segev and Gunadhi | Explicit/timestamp sort | Surrogate attribute and 1TNF |
| Time-join | Segev and Gunadhi | Timestamp sort | None |
| TEJ-3 | Segev and Gunadhi | Timestamp sort | None |
| TEJ-4 | Segev and Gunadhi | Timestamp sort | Surrogate attribute/access chain |
| Several | Leung and Muntz | Timestamp sort | None |
| Interval | Arge et al. | Timestamp sort | None |
| Two | Pfoser and Jensen | Timestamp sort | Partitioned relation; time-ordered updates |
| – | Soo et al. | Timestamp partition | None |
| – | Sitzmann and Stuckey | Timestamp partition | Requires preexisting temporal histogram |
| – | Lu et al. | Timestamp partition | Disallows future time; uses spatial index |

### *3.4 Summary*

We have surveyed temporal join algorithms and proposed a taxonomy of such algorithms. The taxonomy was developed by adapting well-established relational query evaluation paradigms to the temporal operations.

Table 3 summarizes how each temporal join operation proposed in previous work is classified in the taxonomy. We believe that the framework is complete since, disregarding data-model-specific considerations, all previous work naturally fits into one of the proposed categories.

One important property of an algorithm is whether it delivers a partial answer before the entire input is read. Among the algorithms listed in Table 3, only the nested-loop algorithm has this property. Partition-based algorithms have to scan the whole input relation to get the partitions. Similarly, sort-based algorithms have to read the entire input to sort the relation. We note, however, that it is possible to modify the temporal sort-based algorithms to be nonblocking, using the approach of *progressive merge join* [10].

## 4 Engineering the algorithms

As noted in the previous section, an adequate empirical investigation of the performance of temporal join algorithms has not been performed. We concentrate on the temporal equijoin, defined in Sect. 2.4. This join and the related temporal natural join are needed to reconstruct normalized temporal relations [25]. To perform a study of implementations of this join, we must first provide state-of-the-art implementations of the 19 different types of algorithms outlined for this join. In this section, we discuss our implementation choices.

### *4.1 Nested-loop algorithm*

We implemented a simple block-oriented nested-loop algorithm. Each block of the outer relation is read in turn into memory. The outer block is sorted by the explicit joining attribute (actually, pointers are sorted to avoid copying of tuples). Each block of the inner relation is then brought into memory. For a given inner block, each tuple in that block is joined by binary searching the sorted tuples.

This algorithm is simpler than the nested-loop algorithm, EJ-2, described in Sect. 3.3.1 [20,47]. In particular, our algorithm computes only the valid-time equijoin, while EJ-2 computes the valid-time outerjoin, which includes the equijoin in the form of the valid-time natural join. However, our algorithm supports a more general equijoin condition than EJ-2 in that we support matching on any explicit attribute rather than solely on a designated surrogate attribute.

### *4.2 Sort-merge-based algorithms*

We were careful to use a high-performance sort-merge algorithm with the features covered next.

#### 4.2.1 Combining last sort step with merge step

Sort-merge join uses a disk-based sorting phase that starts by generating many small, fully sorted *runs*, then repeatedly merges these into increasingly longer runs until a single run is obtained (this is done for the left-hand side and right-hand side independently). Each step of the sort phase reads and writes the entire relation. The merge phase then scans the fully sorted left-hand and right-hand relations to produce the output relation. A common optimization is to stop the sorting phase one step early, when there is a small number of fully sorted runs. The final step is done in parallel with the merge phase of the join, thereby avoiding one read and one write scan. Our sort-merge algorithms implemented for the performance analysis are based on this optimization. We generated initial runs using an in-memory quicksort on the explicit attributes (ES and ES-H), the timestamp attributes (TS and TS-H), or both (ETS and ETS-H) and then merged the two relations on multiple runs.

#### 4.2.2 Efficient skew handling

As noted in Sect. 3.2.2, sort-merge join algorithms become complicated when the join attributes are not key attributes. Our previous work on conventional joins [30] shows that *intrinsic skew* is generally present in this situation. Even a small amount of intrinsic skew can result in a significant performance hit because the naive approach to handling skew is to

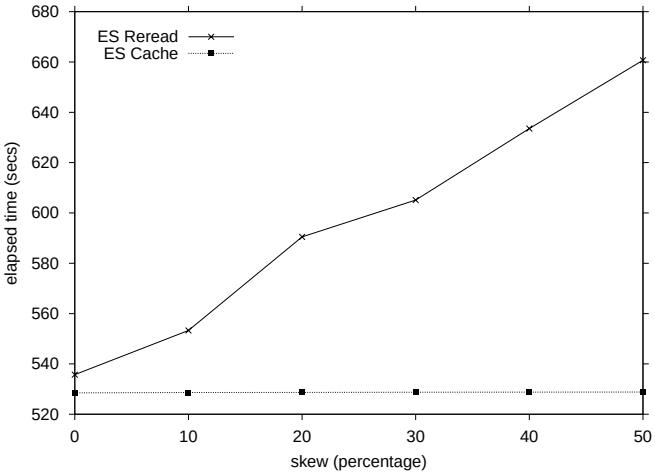


**Fig. 8.** Performance improvement of ES with spooled cache on skewed data

reread the previous tuples in the same *value packet* (containing the identical values for the equijoin attribute); this rereading involves additional I/O operations. We previously proposed several techniques to handle skew efficiently [30]. Among them, SC-$n$ (spooled cache on multiple runs) was recommended due to its strikingly better performance in the presence of skew for both conventional and band joins. This algorithm also exhibits virtually identical performance as a traditional sort-merge join in the absence of skew. SC-$n$ uses a small cache to hold the skewed tuples from the right-hand relation that satisfy the join condition. At the cache's overflow point, the cache data are spooled to disk.

Skew is prevalent in temporal joins. SC-$n$ can be adapted for temporal joins by adding a supplemental predicate (requiring that the tuples overlap) and calculating the resulting timestamps, by intersection. We adopt this spooled cache in ES instead of rereading the previous tuples. The advantage of using spooled cache is shown in Fig. 8. ES Reread is the multirun version of the `explicitSortMerge` algorithm exhibited in Sect. 3.2.2, which backs up the right-hand relation when a duplicate value is found in the left-hand relation.

The two algorithms were executed in the TimeIt system. The parameters are the same as those that will be used in Sect. 5.1. In this experiment, the memory size was fixed at 8 MB and the cache size at 32 KB. The relations were generated with different percentages of *smooth skew* on the explicit attribute. A relation has 1% smooth skew when 1% of the tuples in the relation have one duplicate value on the join attribute and the remaining 98% of the tuples have no duplicates. Since the cache can hold the skewed tuples in memory, no additional I/O is caused by backing up the relation. The performance improvement of using a cache is approximately 25% when the data have 50% smooth skew. We thus use a spooled cache to handle skew. Spooling will generally not occur but is available in case a large value packet is present.

#### 4.2.3 Time-varying value packets and optimized prediction rule

ES utilizes a *prediction rule* to judge if skew is present. (Recall that skew occurs if the two tuples have the same join attribute value.) The prediction rule works as follows. When the last tuple in the right-hand relation (RHR) buffer is visited, the last tuple in the left-hand relation (LHR) buffer is checked to determine if skew is present and the current RHR value packet needs to be put into the cache.

We also implemented an algorithm (TS) that sorts the input relations by start time rather than by the explicit join attribute. Here the RHR value packet associated with a specific LHR tuple is not composed of those RHR tuples with the same start time but rather of those RHR tuples that overlap with the interval of the LHR tuple. Hence value packets are not disjoint, and they grow and shrink as one scans the LHR. In particular, TS puts into the cache only those tuples that *could* overlap in the future: the tuples that do not stop too early, that is, before subsequent LHR tuples start. For an individual LHR tuple, the RHR value packet starts with the first tuple that stops sometime during the LHR tuple's interval and goes through the first RHR tuple that starts after the LHR tuple stops. Value packets are also not totally ordered when sorting by start time.

These considerations suggest that we change the prediction rule in TS. When the RHR reaches a block boundary, the maximum stop time in the current value packet is compared with the start time of the last tuple in the LHR buffer. If the maximum stop time of the RHR value packet is less than the last start time of the LHR, none of the tuples in the value packet will overlap with the subsequent LHR tuples. Thus there is no need to put them in the cache. Otherwise, the value packet is scanned and only those tuples with a stop time greater than the last start time of the LHR are put into the cache, thereby minimizing the utilization of the cache and thus the possibility of cache overflow.

ETS sorts the input relations by explicit attribute first and then by start time. Here the RHR value packet associated with a left tuple is composed of those right tuples that not only have the same value of the explicit attribute but also overlap with the interval of the left tuple. The prediction rules used in ES and TS are combined to decide whether or not to put a tuple or a value packet into the cache.

To make our work complete, we also implemented TES, which sorts the input relations primarily by start time and secondarily by the explicit attribute. The logic of TES is exactly the same as that of TS for the joining phase. We expect the extra sorting by explicit attribute will not help to optimize the algorithm but rather will simply increase the CPU time.

#### 4.2.4 Specialized cache purging

Since the cache size is small, it could be filled up if a value packet is very large or if several value packets accumulate in the cache. For the former, nothing but spooling the cache can be done. However, purging the cache periodically can avoid unnecessary cache spool for the latter and may result in fewer I/O operations.

Purging the cache costs more in TS since the RHR value packets are not disjoint, while in ES they are disjoint both in each run and in the cache. The cache purging process in ES scans the cache from the beginning and stops whenever the first tuple that belongs to the current value packet is met. But in TS, this purging stage cannot stop until the whole cache has been scanned because the tuples belonging to the current value packet are spread across the cache. An inner long-lived

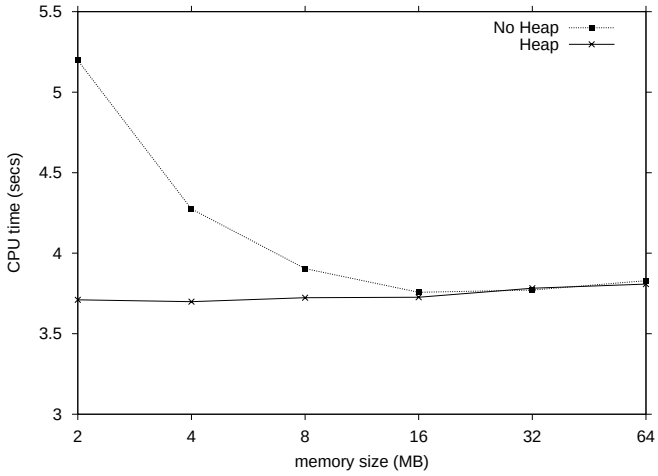


**Fig. 9.** Performance improvement of using a heap in ES

tuple could be kept in the cache for a long time because its time interval could intersect with many LHR tuples.

#### 4.2.5 Using a heap

As stated in Sect. 4.2.1, the final step of sorting is done in parallel with the merging stage. Assuming the two relations are sorted in ascending order, in the merging stage the algorithm first has to find the smallest value from the multiple sorted runs of each relation and then compare the two values to see if they can be joined. The simplest way to find the smallest value is to scan the current value of each run. If the relation is divided into $m$ runs, the cost of selecting the smallest value is $O(m)$. A more efficient way to do this is to use a heap to select the smallest value. The cost of using a heap is $O(log_2 m)$ when $m > 1$. By utilizing a heap, the time complexity is reduced.

At the beginning of the merging step, the heap is built based on the value of the first tuple in each run. Whenever `advance` is called, the run currently on the top of the heap advances its reading pointer to the next tuple. Since the key value of this tuple is no less than the tuple in the current state, it should be propagated down to maintain the heap structure. When a run is backed up, its reading pointer is restored to point to a previously visited tuple, which has a smaller key value, and thus should be propagated up the heap.

When the memory size is relatively small, which indicates that the size of each run is small and therefore that a relation has to be divided into more runs (the number of runs $m$ is large), the performance of using a heap will be much better than that without a heap. However, using a heap causes some pointer swaps when sifting down or propagating up a tuple in the heap, which are not needed in the simple algorithm. When the memory size is sufficiently large, the performance of using a heap will be close to or even worse than that of the simple algorithm.

Figure 9 shows the total CPU time of ES when using and not using a heap. The data used in Fig. 9 are two 64-MB relations. The input relations are joined while using different sizes of memory. Note that the CPU time includes the time of both the sorting step and the merging step. As expected, the performance of using a heap is better than that without a heap when the memory is small. The performance improvement is roughly 40% when the memory size is 2 MB. The performance difference decreases as the memory increases. When the memory size is greater than 32 MB, which is one half of the relation size, using a heap has no benefit. Since using a heap significantly improves the performance when the memory is relatively small and barely degrades performance when the memory is large, we use a heap in all sort-based algorithms.

#### 4.2.6 GRACE and hybrid variants

We implemented both GRACE and hybrid versions of each sort-based algorithm. In the GRACE variants, all the sorted runs of a relation are written to disk before the merging stage. The hybrid variants keep most of the last run of the outer relation in memory. This guarantees that one (multiblock) disk read and one disk write of the memory-resident part will be saved. When the available memory is slightly smaller than the dataset, the hybrid algorithms will require relatively fewer I/O operations.

#### 4.2.7 Adapting the interval join

We consider the interval join a variant of the timestamp sort-merge algorithm (TS). In this paper, we call it TSI and its hybrid variant TSI-H. To be fair, we do not assume the input relations are sorted into one list. Instead, TSI begins with sorting as its first step. Then it combines the last step of the sort with the merge step. The two active lists are essentially two spooled caches, one for each relation. Each cache has the same size as that in TS. This is different from the strategy of keeping a single block of each list in the original paper. A small cache can save more memory for the input buffer, thus reducing the random reads. However, it will cause more cache spools when skew is present. Since timestamp algorithms tend to encounter skew, we choose a cache size that is the same as that in TS, rather than one block.

### *4.3 Partition-based algorithms*

Several engineering considerations also occur when implementing the partition-based algorithms.

#### 4.3.1 Partitioning details

The details of algorithm TP are described elsewhere [54]. We changed TP to use a slightly larger input buffer (32 KB) and a cache for the inner relation (also 32 KB) instead of using a one-page buffer and cache. The rest of the available main memory is used for the outer relation. There is a tradeoff between a large outer input buffer and a large inner input buffer and cache. A large outer input buffer implies a large partition size, which results in fewer seeks for both relations. But the cache is more likely to spool. On the other hand, allocating a large cache and a large inner input buffer results in a smaller outer input buffer, thus a smaller partition size. This will increase random I/O. We chose 32 KB instead of 1 KB (the page size)

as a compromise. The identification of the best cache size is given in Sect. 6 as a direction of future research.

The algorithms ETP and TEP partition the input relations in two steps. ETP partitions the relations by explicit attribute first. For each pair of the buckets to be joined, if none of them fits in memory, a further partition by timestamp attribute will be made to these buckets to increase the possibility that the resulting buckets do not overflow the available buffer space. TEP is similar to ETP, except that it partitions the relations in the reverse order, first by timestamp and then, if necessary, by explicit attribute.

#### 4.3.2 Joining the partitions

The partition-based algorithms perform their second phase, the joining of corresponding partitions of the outer and inner relations, as follows. The outer partition is fetched into memory, assuming that it will not overflow the available buffer space, and pointers to the outer tuples are sorted using an in-memory quicksort. The inner partition is then scanned, using all memory pages not occupied by the outer partition. For each inner tuple, matching outer tuples are found by binary search. If the outer partitions overflow the available buffer space, then the algorithms default to an explicit attribute sort-merge join of the corresponding partitions.

#### 4.3.3 GRACE and hybrid variants

In addition to the conventional GRACE algorithm, we implemented the hybrid buffer management for each partition-based algorithm. In the hybrid algorithms, one outer bucket is designated as memory-resident. Its buffer space is increased accordingly to hold the whole bucket in memory. When the inner relation is partitioned, the inner tuples that map to the corresponding bucket are joined with the tuples in the memory-resident bucket. This eliminates the I/O operations to write and read one bucket of tuples for both the inner and the outer relation. Similar to the hybrid sort-based algorithms, the hybrid partition-based algorithms are supposed to have better performance when the input relation is slightly larger than the available memory size.

### *4.4 Supporting the iterator interface*

Most commercial systems implement the relational operators as *iterators* [18]. In this model, each operator is realized by three procedures called *open*, *next*, and *close*. The algorithms we investigate in this paper can be redesigned to support the iterator interface.

The nested-loop algorithm and the explicit partitioning algorithms are essentially the corresponding snapshot join algorithms except that a supplemental predicate (requiring that the tuples overlap) and the calculation of the resulting timestamps are added in the *next* procedure.

The timestamp partitioning algorithms determine the periods for partitions by sampling the outer relation and partition the input relations in the *open* procedure. The *next* procedure calls the *next* procedure of nested-loop join for each pair of partitions. An additional predicate is added in the *next* procedure to determine if a tuple should be put into the cache.

The sort-based algorithms generate the initial sorted runs for the input relations and merge runs until the final merge step is left in the *open* procedure. In the *next* procedure, the inner runs, the cache, and the outer runs are scanned to find a match. At the same time, the inner tuple is examined to decide whether to put it in the cache. The *close* procedure destroys the input runs and deallocates the cache. The *open* and *close* procedures of interval join algorithms are the same as the other sort-based algorithms. The *next* procedure gets the next tuple from the sorted runs and scans the cache to find the matching tuple and purges the cache at the same time.

## 5 Performance

We implemented all 19 algorithms enumerated in Table 2 and tested their performance under a variety of data distributions, including skewed explicit and timestamp distributions, timestamp durations, memory allocations, and database sizes. We ensured that all algorithms generated exactly the same output tuples in all of the experiments (the ordering of the tuples will differ).

The remainder of this section is organized as follows. We first give details on the join algorithms used in the experiments and then describe the parameters used in the experiments. Sections 5.2 to 5.9 contain the actual results of the experiments. Section 5.10 summarizes the results of the experiments.

### *5.1 Experimental setup*

The experiments were developed and executed using the TimeIT [17] system, a software package supporting the prototyping of temporal database components. Using TimeIT, we fixed several parameters describing all test relations used in the experiments. These parameters and their values are shown in Table 4. In all experiments, tuples were 16 bytes long and consisted of two explicit attributes, both being integers and occupying 4 bytes, and two integer timestamps, each also requiring 4 bytes. Only one of the explicit attributes was used as the joining attribute. This yields result tuples that are 24 bytes long, consisting of 16 bytes of explicit attributes from each input tuple and 8 bytes for the timestamps.

We fixed the relation size at 64 MB, giving four million tuples per relation. We were less interested in absolute relation size than in the ratio of input size to available main memory. Similarly, the ratio of the page size to the main memory size and the relation size is more relevant than the absolute page size. A scaling of these factors would provide similar results. In all cases, the generated relations were randomly ordered with respect to both their explicit and timestamp attributes.

The metrics used for all experiments are listed in Table 5. In a modern computer system, a random disk access takes about 10 ms, whereas accessing a main memory location typically takes less than 60 ns [42]. It is reasonable that a sequential I/O takes about one tenth the time of a random I/O. Modern computer systems usually have hardware data cache, which can reduce the CPU time on cache hit. Therefore, we chose the join attribute compare time as 20 ns, which was slightly less

**Table 4.** System characteristics

| *Parameter* | *Value* |
|---|---|
| Relation size | 64 MB |
| Tuple size | 16 bytes |
| Tuples per relation | 4 million |
| Timestamp size ($[s,e]$) | 8 bytes |
| Explicit attribute size | 8 bytes |
| Relation lifespan | 1,000,000 chronons |
| Page size | 1 KB |
| Output buffer size | 32 KB |
| Cache size in sort-merge | 64 KB |
| Cache size in partitioning | 32 KB |

**Table 5.** Cost metrics

| *Parameter* | *Value* |
|---|---|
| Sequential I/O cost | 1 ms |
| Random I/O cost | 10 ms |
| Join attribute compare | 20 ns |
| Timestamp compare | 20 ns |
| Pointer compare | 20 ns |
| Pointer swap | 60 ns |
| Tuple move | 80 ns |

than, while in the same magnitude of, the memory access time. The cost metrics we used is the average memory access time given a high hit ratio ($> 90\%$) of cache. It is possible that the CPU cache has lower hit ratio when running some algorithms. However, the magnitude of the memory access time will not change. We assumed that the sizes of both a timestamp and a pointer were the same as the size of an integer. Thus, their compare times are the same as that of the join attribute. A pointer swap takes three times as long as the pointer compare time because it needs to access three pointers. A tuple move takes four times as long as the integer compare time since the size of a tuple is four times that of an integer.

We measured both main memory operations and disk I/O operations. To eliminate any undesired system effects from the results, all operations were counted using facilities provided by TimeIT. For disk operations, random and sequential access was measured separately. We included the cost of writing the output relation in the experiments since sort-based and partition-based algorithms exhibit dual random and sequential I/O patterns when sorting/coalescing and partitioning/merging. The total time was then computed by weighing each parameter by the time values listed in Table 5.

Table 6 summarizes the values of the system parameters that varied among the different experiments. Each row of the table identifies the figures that illustrate the results of the experiments given the parameters for the experiment. The reader may have the impression that the intervals are so small that they are almost like the standard equijoin atributes. Are there tuples overlapping each other? In many cases, we performed the self-join, which guaranteed for each tuple in one relation that there is at least one matching tuple in the other relation. Long-duration timestamps (100 chronons) were used in two experiments. It guaranteed that there were on average four tuples valid in each chronon. Two other experiments examine the case where one relation has short-duration timestamps and the other has long-duration timestamps. Therefore, our experiments actually examined different degrees of overlapping.

### *5.2 Simple experiments*

In this section, we perform three "base case" experiments, where the join selectivity is low, i.e., for an equijoin of valid-time relations $r$ and $s$, a given tuple $x \in r$ joins with one, or few, tuples $y \in s$. The experiments incorporate random data distributions in the explicit join attributes and short and long time intervals in the timestamp attributes.

#### 5.2.1 Low explicit selectivity with short timestamps

In this experiment, we generated a relation with little explicit matching and little overlap and joined the relation with itself. This mimics a foreign key-primary key natural join in that the cardinality of the result is the same as one of the input relations. The relation size was fixed at 64 MB, corresponding to four million tuples. The explicit joining attribute values were integers drawn from a space of $2^{31} - 1$ values. For the given cardinality, a particular explicit attribute value appeared on average in only one tuple in the relation. The starting timestamp attribute values were randomly distributed over the relation lifespan, and the duration of the interval associated with each tuple was set to one chronon. We ran each of the 19 algorithms using the generated relation, increasing main memory allocations from 2 MB, a 1:32 memory to input size ratio, to 64 MB, a 1:1 ratio.

The results of the experiment are shown in Fig. 10. In each panel, the ordering of the legend corresponds to the order of either the rightmost points or the leftmost points of each curve. The actual values of each curve in all the figures may be found in the appendix of the associated technical report [16]. Note that both the $x$-axis and the $y$-axis are log-scaled. As suspected, nested loop is clearly not competitive. The general nested-loop algorithm performs very poorly in all cases but the highest memory allocation. At the smallest memory allocation, the least expensive algorithm, EP, enjoys an 88% performance increase. Only at the highest memory allocation, that is, when the entire left-hand side relation can fit in main memory, does the nested-loop algorithm have comparable performance with other algorithms. Given the disparity in performance and given that various characteristics, such as skew or the presence of long-duration tuples, do not impact the performance of the nested-loop algorithm, we will not consider this algorithm in the remainder of this section.

To get a better picture of the performance of the remaining algorithms, we plot them separately in Fig. 11. From this figure on, we eliminate the noncompetitive nested loop. We group the algorithms that have a similar performance and retain only a representative curve for each group in the figures. In this figure, TES-H and TSI-H have performances very similar to that of TS-H; ETS-H has a performance very similar to that of ES-H; ETP-H, TP-H, and TEP-H have performances similar to that of EP-H; the remaining algorithms all have a performance similar to that of EP.

In this graph, only the $x$-axis is log-scaled. The sort-based and partition-based algorithms exhibit largely the same performance, and the hybrid algorithms outperform their GRACE

**Table 6.** Experiment parameters

| Figure numbers | *Explicit skew* (%) | *Timestamp skew* (%) | *Timestamp duration* (chronons) | *Outer size* (MB) | *Inner size* (MB) | *Memory size* (MB) |
|---|---|---|---|---|---|---|
| 10 and 11 | None | None | 1 | 64 | 64 | 2–64 |
| 12 | None | None | 100 | 64 | 64 | 2–64 |
| 13 | None | None | Outer: 1; Inner: 100 | 64 | 64 | 2–64 |
| 14 | None | None | 1 | 4–64 | 64 | 16 |
| 15 and 16 | None | None | 100 | 4–64 | 64 | 16 |
| 17 | None | None | Outer: 1; Inner: 100 | 4–64 | 64 | 16 |
| 18 | 0–100% one side | None | 1 | 64 | 64 | 16 |
| 19 | None | 0–100% one side | 1 | 64 | 64 | 16 |
| 20 | 0–100% one side | 0–100% one side | 1 | 64 | 64 | 16 |
| 21 | 0–4% both sides | None | 1 | 64 | 64 | 16 |
| 23 | None | 0–4% both sides | 1 | 64 | 64 | 16 |
| 25 | 0–4% both sides | 0–4% both sides | 1 | 64 | 64 | 16 |

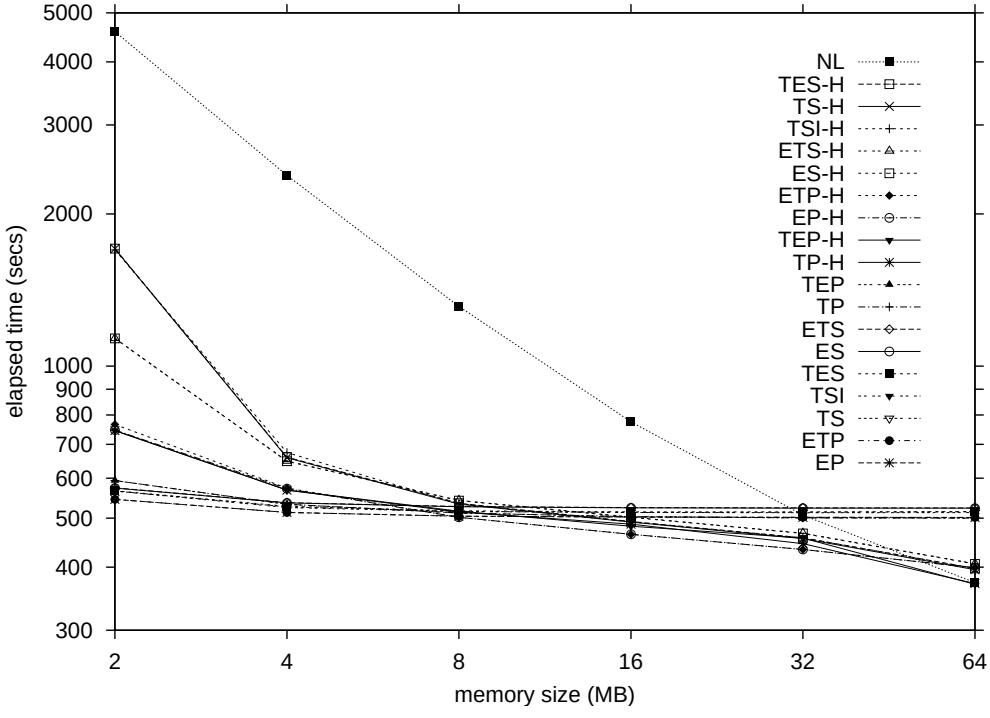


**Fig. 10.** Low explicit selectivity, low timestamp selectivity

counterparts at high memory allocations, in this case when the ratio of main memory to input size reaches approximately 1:8 (2 MB of main memory) or 1:4 (4 MB of main memory). The poor performance of the hybrid algorithms stems from reserving buffer space to hold the resident run/partition, which takes buffer space away from the remaining runs/partitions, causing the algorithms to incur more random I/O. At small memory allocations, the problem is acute. Therefore, the hybrid group starts from a higher position and ends in a lower position. The GRACE group behaves in the opposite way.

The performance differences between the sort-based algorithms and their partitioning counterparts are small, and there is no absolute winner. TES, the sort-merge algorithm that sorts the input relation primarily by start time and secondarily by explicit attribute, has a slightly worse performance than TS, which sorts the input relation by start time only. Since the order of the start time is not the order of the time interval, the extra sorting by explicit attribute does not help in the merging step. The program logic is the same as for TS, except for the extra sorting. We expect TES will always perform a little worse than TS. Therefore, neither TES nor TES-H will be considered in the remainder of this section.

### 5.2.2 Long-duration timestamps

In the experiment described in the previous section, the join selectivity was low since explicit attribute values were shared among few tuples and tuples were timestamped with intervals of short duration. We repeated the experiment using long-duration timestamps. The duration of each tuple timestamp was fixed at 100 chronons, and the starting timestamps were randomly distributed throughout the relation lifespan. As before, the explicit join attribute values were randomly dis-

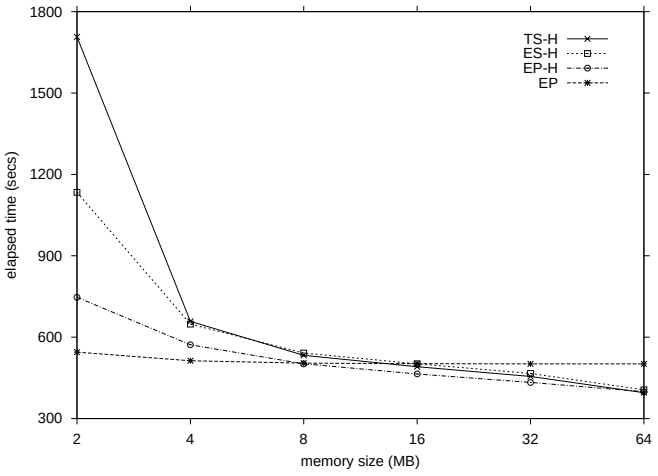


**Fig. 11.** Low explicit selectivity, low timestamp selectivity (without nested loop)

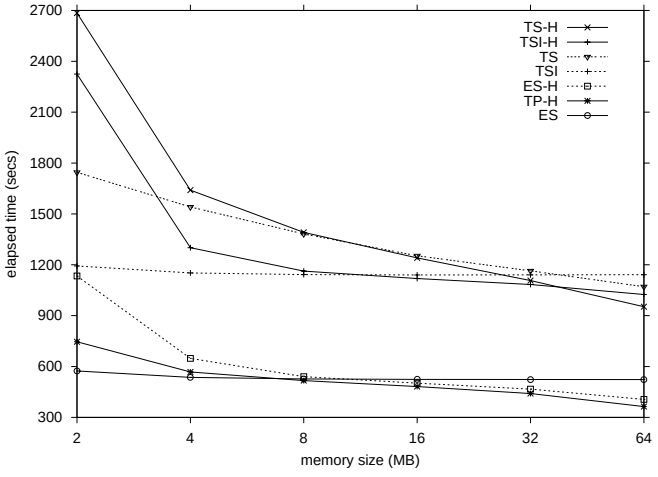


**Fig. 12.** Low explicit selectivity (long-duration timestamps)

tributed integers; thus the size of the result was just slightly larger due to the long-duration timestamps.

The results are shown in Fig. 12, where the $x$-axis is log-scaled. In this figure, the group of ES-H and ETS-H are represented by ES-H; the group of ETP-H, EP-H, TEP-H, and TP-H by TP-H; the group of TP, TEP, ES, ETS, EP, and ETP by ES; and the rest are retained. The timestamp sorting algorithms, TS and TS-H, suffer badly. Here, the long duration of the tuple lifespans did not cause overflow of the tuple cache used in these algorithms. To see this, recall that our input relation cardinality was four million tuples. For a 1,000,000 chronon relation lifespan, this implies that $4,000,000/1,000,000 = 4$ tuples arrive per chronon. Since tuple lifespans were fixed at 100 chronons, it follows that $4 \times 100 = 400$ tuples should be scanned before any purging of the tuple cache can occur. However, a 64-KB tuple cache, capable of holding 4000 tuples, does not tend to overflow. Detailed examination verified that the cache never overflowed in these experiments. The poor performance of TS and TS-H are caused by the repeated in-memory processing of the long-lived tuples.

TSI and TSI-H also suffer in the case of long duration but are better than TS and TS-H when the main memory size is small. TSI improves the performance of TS by 32% at the smallest memory allocation, while TSI-H improves the performance of TS-H by 13%. Our detailed results show that the TS had slightly less I/O time than TSI. TS also saved some time in tuple moving since it did not move every tuple into cache. However, it spent much more time in timestamp comparing and pointer moving. In TSI, each tuple only joined with the tuples in the cache of the other relation. The caches in TSI were purged during the join process; thus the number of timestamp comparisons needed by the next tuple was reduced. In TS, an outer tuple joined with both cache tuples and tuples in the input buffer of the inner relation, and the input buffer was never purged. Therefore, TS had to compare more timestamps. Pointer moving is needed in the heap maintenance, which is used to sort the current tuples in each run. TS frequently backed up the inner runs inside the inner buffer and scanned tuples in the value packets multiple times. In each scan, the heap for the inner runs had to sort the current inner tuples again. In TSI, the tuples are sorted once and kept in order in the caches. Therefore, the heap overhead is small. When the main memory size is small, the number of runs is large, as are the heap size and the heap overhead.

The timestamp partitioning algorithms, TP and TP-H, have a performance very similar to that described in Sect. 5.2.1. There are two main causes of the good performance of TP and TP-H. The first is that TP does not replicate long-lived tuples that overlap with multiple partition intervals. Otherwise, TP would need more I/O for the replicated tuples. The second is that TP sorts each partition by the explicit attribute. The long duration does not have any effect on the performance of the in-memory joining. All the other algorithms sort or partition the relations by explicit attributes. Therefore, their performance is not affected by the long duration.

We may conclude from this experiment that the timestamp sort-based algorithms are quite sensitive to the durations of input tuple intervals. When tuple durations are long, the in-memory join in TS and TS-H performs poorly due to the need to repeatedly back up the tuple pointers.

### 5.2.3 Short- and long-duration timestamps

In the experiments described in the previous two sections, the timestamps are either short or long for both relations. We examined the case where the durations for the two input relations are different. The duration of each tuple timestamp in the outer relation was fixed at 1 chronon, while the duration of that in the inner relation was fixed at 100 chronons. We carefully generated the two relations so that the outer relation and the inner relation had a one-to-one relationship. For each tuple in the outer relation, there is one tuple in the inner relation that has the same value of the explicit attributes and the same start time as the outer tuple with a long duration instead of a short duration. This guaranteed that the selectivity was between that of the two previous experiments. As before, the explicit join attribute values and the start time were randomly distributed.

The results are shown in Fig. 13, where the $x$-axis is log-scaled. The groups of the curves are the same as in Fig. 12. The relative positions of the curves are similar to those in the long-duration experiment. The performance of the timestamp sorting algorithms were even worse than that of the others, but better than that in the experiment where long-duration tu-

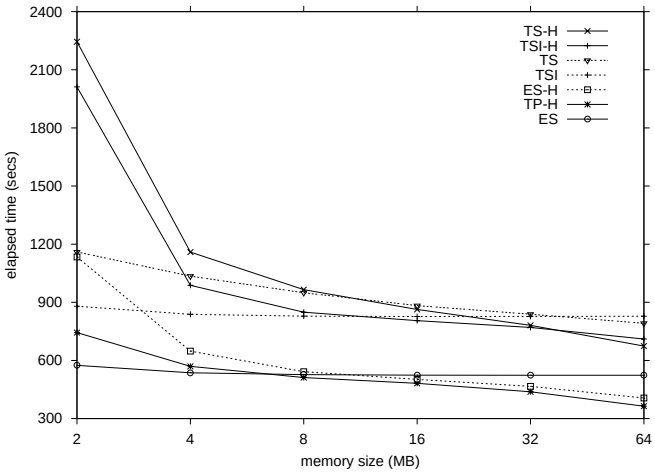


**Fig. 13.** Low explicit selectivity (short-duration timestamps join long-duration timestamps)

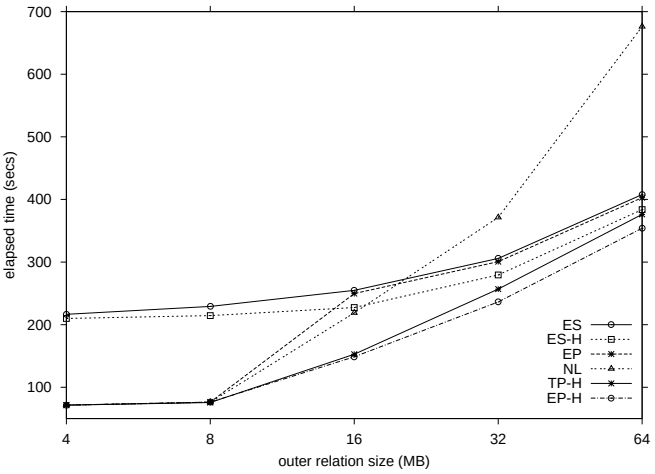


**Fig. 14.** Different relation sizes (short-duration timestamps)

ples were in both input relations. Long-duration tuples reduce the size of value packets for each tuple on only one side and therefore result in fewer timestamp comparisons in all four timestamp sorting algorithms and fewer backups in TS and TS-H.

We also exchanged the outer and inner relations for this experiment and observed results identical to those in Fig. 13. This indicates that whether the long-duration tuples exist in the outer relation or the inner relation has little impact on the performance of any algorithm.

### *5.3 Varying relation sizes*

It has been shown for snapshot join algorithms that the relative sizes of the input relations can greatly affect which sort- or partition-based strategy is best [18]. We investigated this phenomenon in the context of valid-time databases.

We generated a series of relations, increasing in size from 4 MB to 64 MB, and joined them with a 64-MB relation. The memory allocation used in all trials was 16 MB, the size at which all algorithms performed most closely in Fig. 11. As in the previous experiments, the explicit join attribute values in all relations were randomly distributed integers. Short-duration timestamps were used to mitigate the in-memory effects on TS and TS-H seen in Fig. 12. As before, starting timestamps were randomly distributed over the relation lifespan. Since the nested-loop algorithm is expected to be a competitor when one of the relations fits in the memory, we incorporated this algorithm into this experiment. The results of the experiment are shown in Fig. 14. In this figure, ES represents all the GRACE sorting algorithms, ES-H all the hybrid sorting algorithms, EP all the GRACE partitioning algorithms, TP-H the hybrid timestamp partitioning algorithms, and EP-H the hybrid explicit partitioning algorithms, and NL is retained.

The impact of a differential in relation sizes for the partition-based algorithms is clear. When an input relation is small relative to the available main memory, the partition-based algorithms use this relation as the outer relation and build an in-memory partition table from it. The inner relation is then linearly scanned, and for each inner tuple the in-memory partition table is probed for matching outer tuples. The benefit of this approach is that each relation is read only once, i.e., no intermediate writing and reading of generated partitions occurs. Indeed, the inner relation is not partitioned at all, further reducing main memory costs in addition to I/O savings.

The nested-loop algorithm has the same I/O costs as partition-based algorithms when one of the input relations fits in the main memory. When the size of the smaller input relation is twice as large as the memory size, the performance of nested-loop algorithms is worse than that of any other algorithms. This is consistent with the results shown in Fig. 10.

An important point to note is that this strategy is beneficial regardless of the distribution of either the explicit join attributes and/or the timestamp attributes, i.e., it is unaffected by either explicit or timestamp skew. Furthermore, no similar optimization is available for sort-based algorithms. Since each input relation must be sorted, both relations must be read and written once to generate sorted runs and subsequently read once to scan and match joining tuples.

To further investigate the effectiveness of this strategy, we repeated the experiment of Fig. 14 with long-duration timestamps, i.e., tuples were timestamped with timestamps 100 chronons in duration. We did not include the nested-loop algorithm because we did not expect the long-duration tuples to have any impact on it. The results are shown in Fig. 15. The grouping of the curves in this figure is slightly different from the grouping in Fig. 14 in that timestamp sorting algorithms are separated instead of grouped together.

As expected, long-duration timestamps adversely affect the performance of all the timestamp sorting algorithms for reasons stated in Sect. 5.2.2. The performance of TSI and TSI-H is slightly better than that of TS and TS-H, respectively. This is consistent with the results at 16 MB memory size in Fig. 12. Replotting the remaining algorithms in Fig. 16 shows that the long-duration timestamps do not significantly impact the efficiency of other algorithms.

In both the short-duration and the long-duration cases, the hybrid partitioning algorithms show the best performance. They save about half of the I/O operations of their GRACE counterparts when the size of the outer relation is 16 MB. This is due to the hybrid strategy.

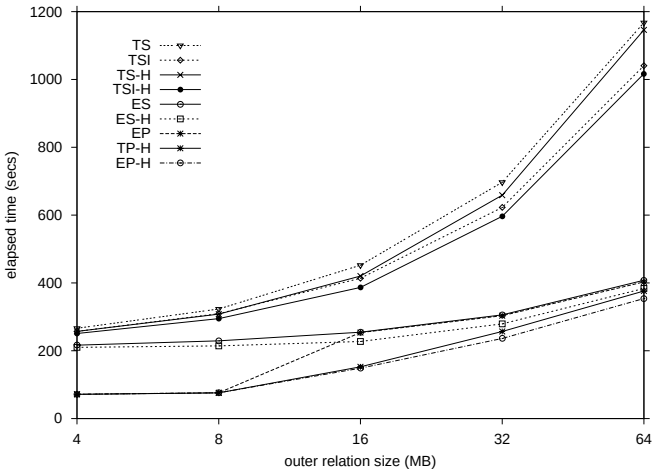


**Fig. 15.** Different relation sizes (long-duration timestamps)

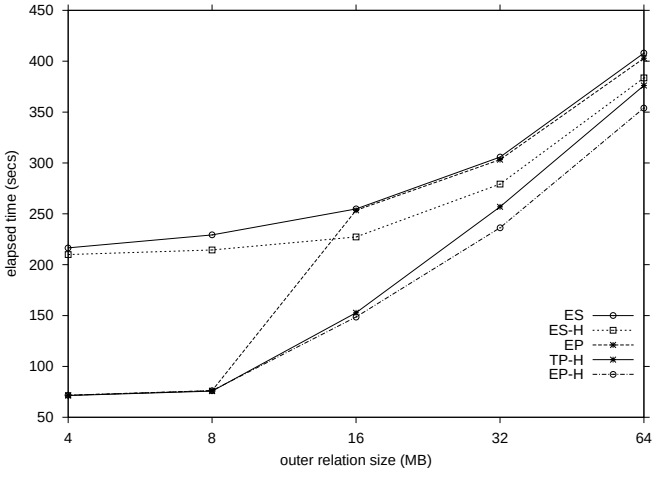


**Fig. 16.** Different relation sizes (long-duration timestamps, without TS/TS-H)

We further changed the input relations so that the tuples in the outer relations have the fixed short duration of 1 chronon and those in the outer relations have the fixed long duration of 100 chronons. Other features of the input relations remain the same. The reults, as shown in Fig. 17, are very similar to the long-duration case. The performance of timestamp sorting algorithms is slightly better than that in Fig. 15. Again, we regenerated the relations such that the tuples in the outer relation have the long-duration fixed at 100 chronons and those in the inner relation have the short-duration fixed at 1 chronon. The results are almost identical to those shown in Fig. 17.

The graph shows that partition-based algorithms should be chosen whenever the size of one or both of the input relations is small relative to the available buffer space. We conjecture that the choice between explicit partitioning and timestamp partitioning is largely dependent on the presence or absence of skew in the explicit and/or timestamp attributes. Explicit and timestamp skew may or may not increase I/O cost; however, they will increase main memory searching costs for the corresponding algorithms, as we now investigate.

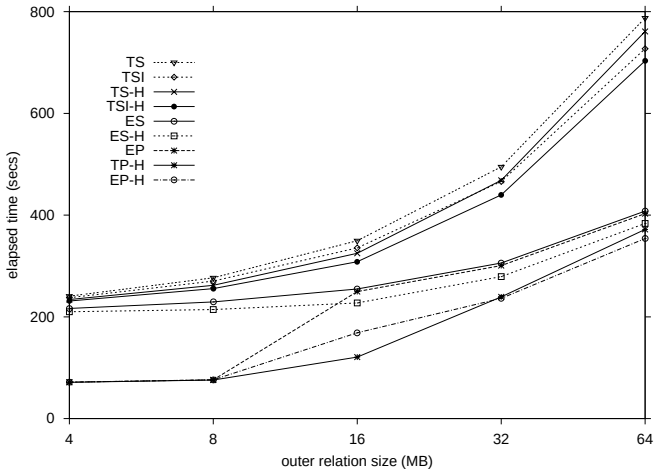


**Fig. 17.** Different relation sizes (short- and long-duration timestamps)

### *5.4 Explicit attribute skew*

As in the experiments described in Sect. 5.3, we fixed the main memory allocation at 16 MB to place all algorithms on a nearly even footing. The inner and outer relation sizes were fixed at 64 MB each. We generated a series of outer relations with increasing explicit attribute skew, from 0% to 100% in 20% increments. Here we generated data with *chunky skew*. The explicit attribute has 20% chunky skew, which indicates that 20% of the tuples in this relation have the same explicit attribute value. Explicit skew was ensured by generating tuples with the same explicit join attribute value. Short-duration timestamps, randomly distributed over the relation lifespan, were used to mitigate the long-duration timestamp effect on timestamp sorting algorithms. The results are shown in Fig. 18. In this figure, TSI, TS, TEP, and TP are represented by TS and their hybrid counterparts by TS-H, and other algorithms are retained.

There are three points to emphasize in this graph. First, the explicit partitioning algorithms, i.e., EP, EP-H, ETP, and ETP-H, show increasing costs as the explicit skew increases. The performance of EP and EP-H degrades dramatically with increasing explicit skew. This is due to the overflowing of main memory partitions, causing subsequent buffer thrashing. The effect, while pronounced, is relatively small since only one of the input relations is skewed. Encountering skew in both relations would exaggerate the effect. Although the performance of ETP and ETP-H also degrades, the changes are much less pronounced. The reason is that they employ time partitioning to reduce the effect of explicit attribute skew.

As expected, the group of algorithms that perform sorting or partitioning on timestamps, TS, TS-H, TP, TP-H, TEP, and TEP-H, have relatively flat performance. By ordering or partitioning by time, these algorithms avoid effects due to explicit attribute distributions.

The explicit sorting algorithms, ES, ES-H, ETS, and ETS-H, perform very well. In fact, the performance of ES and ES-H increases as the skew increases. As the skew increases, by default the relations become increasingly sorted. Hence, ES and ES-H expend less effort during run generation.

We conclude from this experiment that if high explicit skew is present in one input relation, then explicit sorting,

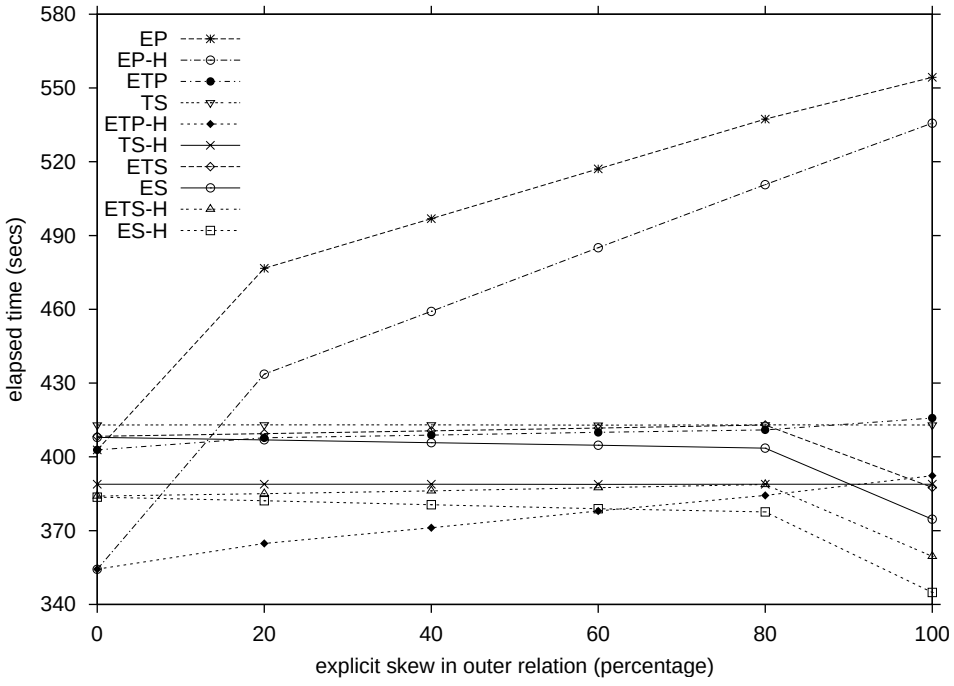


**Fig. 18.** Explicit attribute skew (short-duration timestamps)

timestamp partitioning, and timestamp sorting appear to be the better alternatives. The choice among these is then dependent on the distribution and the length of tuple timestamps, which can increase the amount of timestamp skew present in the input, as we will see in the next experiment.

### *5.5 Timestamp skew*

Like explicit attribute distributions, the distribution of timestamp attribute values can greatly impact the efficiency of the different algorithms. We now describe a study on the effect of this aspect.

As in the experiments described in Sect. 5.3, we fixed the main memory allocation at 16 MB and the sizes of all input relations at 64 MB. We fixed one relation with randomly distributed explicit attributes and randomly distributed tuple timestamps, and we generated a series of relations with increasing timestamp attribute chunky skew, from 0% to 100% in 20% increments. The timestamp attribute has 20% chunky skew, which indicates that 20% of the tuples in the relation are in one value packet. The skew was created by generating tuples with the same interval timestamp. Short-duration timestamps were used in all relations to mitigate the long-duration timestamp effect on timestamp sorting algorithms. Explicit join attribute values were distributed randomly. The results of the experiment are shown in Fig. 19. In this figure, all the GRACE explicit algorithms are represented by EP, hybrid explicit sorting algorithms by ES-H, and hybrid explicit partition algorithms by EP-H; the remaining algorithms are retained.

Four interesting observations may be made. First, as expected, the timestamp partitioning algorithms, i.e., TP, TEP, TP-H, and TEP-H, suffered increasingly poorer performance as the amount of timestamp skew increased. This skew causes overflowing partitions. The performance all four of these algorithms is good when the skew is 100% because TP and TP-H become explicit sort-merge joins and TEP and TEP-H become explicit partition joins. Second, TSI and TSI-H also exhibited poor performance as the timestamp skew increased because 20% skew in the outer relation caused the outer cache to overflow. Third, TS and TS-H show increased performance at the highest skew percentage. This is due to the sortedness of the input, analogous to the behavior of ES and ES-H in the previous experiment. Finally, as expected, the remaining algorithms have flat performance across all trials.

When timestamp skew is present, timestamp partitioning is a poor choice. We expected this result, as it is analogous to the behavior of partition-based algorithms in conventional databases, and similar results have been reported for temporal coalescing. The interval join algorithms are also bad choices when the amount of timestamp skew is large. A small amount of timestamp skew can be handled efficiently by increasing the cache size in interval join algorithms. We will discuss this issue again in Sect. 5.8. Therefore, the two main dangers to good performance are explicit attribute skew and/or timestamp attribute skew. We investigate the effects of simultaneous skew next.

### *5.6 Combined explicit/timestamp attribute skew*

Again, we fixed the main memory allocation at 16 MB and set the input relation sizes at 64 MB. Timestamp durations were set to 1 chronon to mitigate the long-duration timestamp effect on the timestamp sorting algorithms. We then generated a series of relations with increasing explicit and timestamp chunky skew, from 0% to 100% in 20% increments. Skew was created by generating tuples with the same explicit joining attribute value and tuple timestamp. The explicit skew and the timestamp skew are orthogonal. The results are shown in

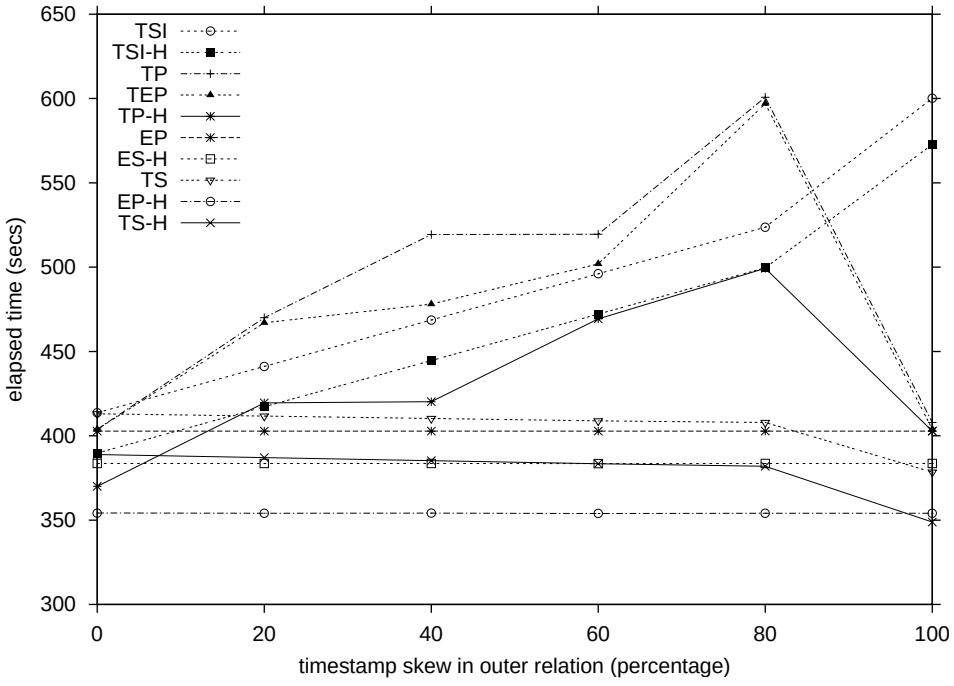


**Fig. 19.** Timestamp attribute skew (short-duration timestamps)

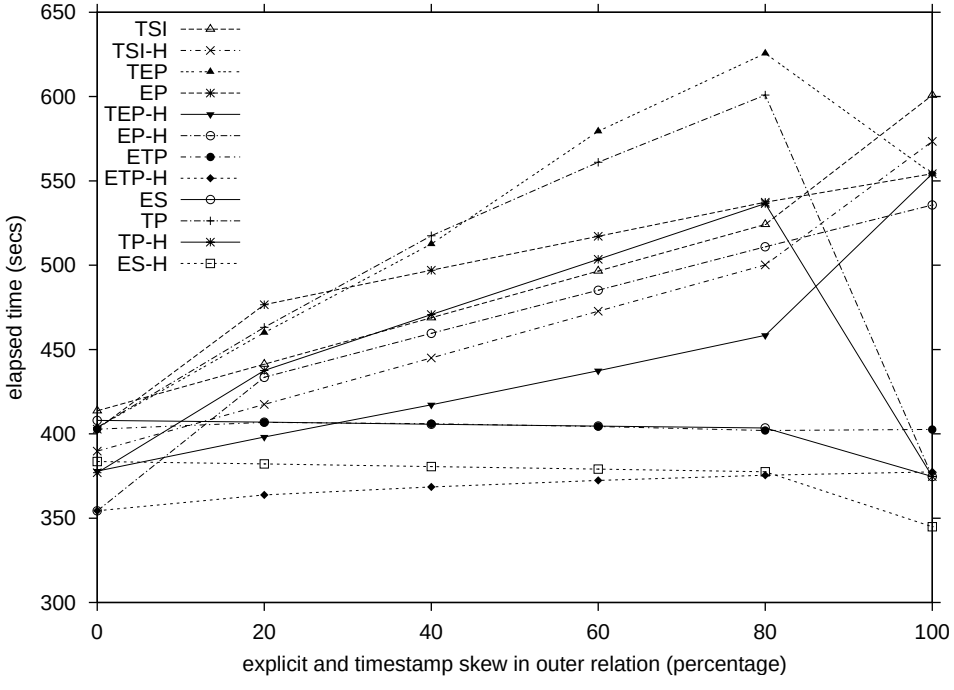


**Fig. 20.** Combined explicit/timestamp attribute skew

Fig. 20. In this figure, ETS, ES, and TS are represented by ES; and ETS-H, ES-H, and TS-H by ES-H; the other algorithms are retained.

The algorithms are divided into three groups in terms of performance. As expected, most of the partition-based algorithms and the interval join algorithms, TEP, TEP-H, TP, TP-H, EP, EP-H, TSI, and TSI-H, show increasingly poorer performance as the explicit and timestamp skew increases. The remaining explicit/timestamp sorting algorithms show relatively flat performance across all trials, and the explicit sorting and timestamp sorting algorithms exhibit increasing performance as the skew increases, analogous to their performance in the experiments described in Sects. 5.4 and 5.5. While the elapsed time of ETP and ETP-H increases slowly along with increasing skew, these two algorithms perform very well. This is analogous to their performance in the experiments described in Sect. 5.4.

### *5.7 Explicit attribute skew in both relations*

In previous work [30], we studied the effect of data skew on the performance of sort-merge joins. There are three types of skew: outer relation skew, inner relation skew, and dual skew.

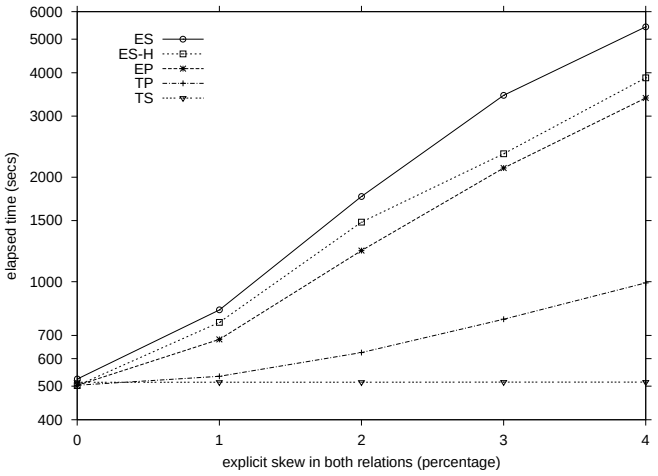


**Fig. 21.** Explicit attribute skew in both relations

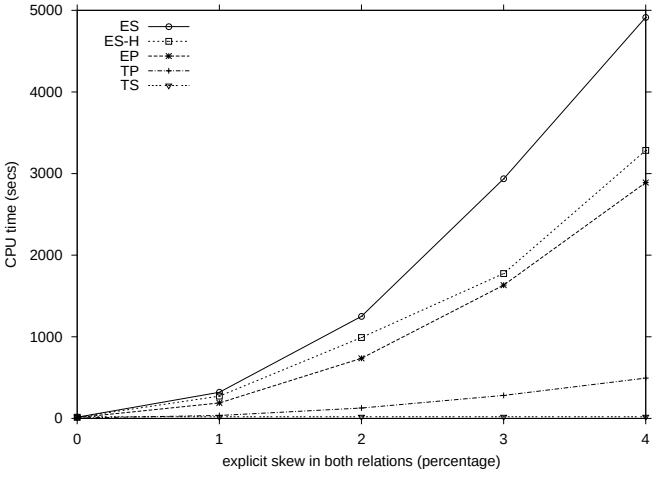


**Fig. 22.** Explicit attribute skew in both relations

Outer skew occurs when value packets in the outer relation cross buffer boundaries. Similarly, inner skew occurs when value packets in the inner relation cross buffer boundaries. Dual skew indicates that outer skew occurs in conjunction with inner skew. While outer skew does not cause any problems for TS and TS-H, it degrades the performance of TSI and TSI-H; dual skew degrades the performance of the TS and TS-H joins. In this section, we compare the performance of the join algorithms in the presence of dual skew in the explicit attribute.

The main memory allocation was fixed at 16 MB and the size of all input relations at 64 MB. We generated a series of relations with increasing explicit attribute chunky skew, from 0% to 4% in 1% increments. To ensure dual skew, we performed self-join on these relations. Short-duration timestamps, randomly distributed over the relation lifespan, were used to mitigate the long-duration timestamp effect on the timestamp sorting algorithms. The results are shown in Fig. 21. In this figure, all the explicit partitioning algorithms are represented by EP, all the timestamp partitioning algorithms by TP, all the sort merge algorithms except ES and ES-H by TS, and ES and ES-H are retained.

There are three points to discuss regarding the graph. First, the explicit algorithms, i.e., ES, ES-H, EP, EP-H, ETP, and ETP-H, suffer when the skew increases. Although the numbers of I/O operations of these algorithms increase along with the increasing skew, the I/O-incurred difference between the highest and the lowest skew is only 2 s. The difference in the output relation size between the highest and the lowest skew is only 460 KB, which leads to about a 4.6-s performance difference. Then what is the real reason for the performance hit of these algorithms? Detailed examination revealed that it is in-memory operations that cause the poor performance of these algorithms. When data skew is present, these algorithms have to do substantial in-memory work to perform the join. This is illustrated in Fig. 22, which shows the CPU time used by each algorithm. To present the difference clearly, we do not use a log-scale $y$-axis. Note that six algorithms, i.e., ETS, TSI, TS, ETS-H, TSI-H, and TS-H, have very low CPU cost (less than 30 s) in all cases. So their performance does not degrade when the degree of skew increases.

Second, the performance of the timestamp partitioning algorithms, i.e., TP, TP-H, TEP, and TEP-H, degrade with increasing skew, but not as badly as do the explicit algorithms. Although timestamp partitioning algorithms sort each partition by the explicit attribute, the explicit attribute inside each partition is not highly skewed. For example, if $n$ tuples have the same value as the explicit attribute, they will be put into one partition after being hashed in EP. In the join phase, there will be an $n \times n$ loop within the join. In TP, this value packet will be distributed evenly across partitions. Assuming there are $m$ partitions, each partition will have $n/m$ of these tuples, which leads to an $n^2/m^2$ loop within the join per partition. The total number of join operations in TP will be $n^2/m$, which is $1/m$ of that of EP. This factor can be seen from Fig. 22.

Finally, the timestamp sorting algorithms, i.e., TS, TS-H, TSI, TSI-H, ETS, and ETS-H, perform very well under explicit skew. TS and TS-H only use the timestamp to determine if a backup is needed. TSI and TSI-H only use the timestamp to determine if the cache tuples should be removed. We see the benefit of the secondary sorting on the timestamp in the algorithms ETS and ETS-H. Since these two algorithms define the value packet by both the explicit attribute and the timestamp, the big loop in the join phase is avoided.

From this experiment, we conclude that when explicit dual skew is present, all the explicit algorithms are poor choices except for ETS and ETS-H. The effects of timestamp dual skew are examined next.

### *5.8 Timestamp dual skew*

Like explicit dual skew, timestamp dual skew can affect the performance of the timestamp sort-merge join algorithms. We look into this effect.

We fixed main memory at 16 MB and input relations at 64 MB. We generated a series of relations with increasing timestamp chunky skew, from 0% to 4% in 1% increments. To ensure dual skew, we performed a self-join on these relations. Short-duration timestamps, randomly distributed over the relation lifespan, were used to mitigate the long-duration timestamp effect on timestamp sorting algorithms. The explicit attribute values were also distributed randomly. The results are shown in Fig. 23. In this figure, GRACE explicit sort merge algorithms are represented by ES; all hybrid partition-

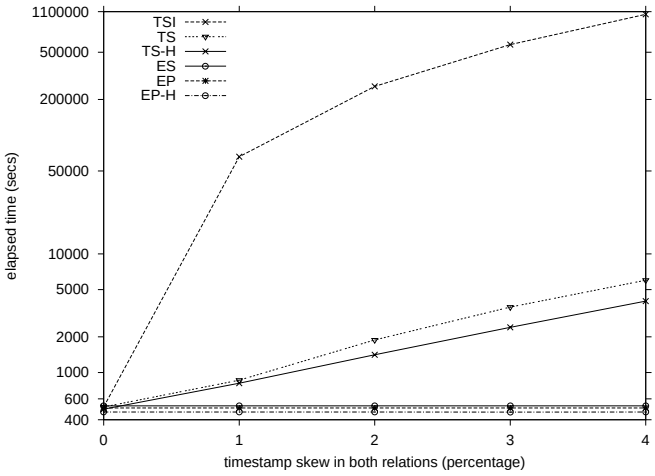


**Fig. 23.** Timestamp attribute skew in both relations

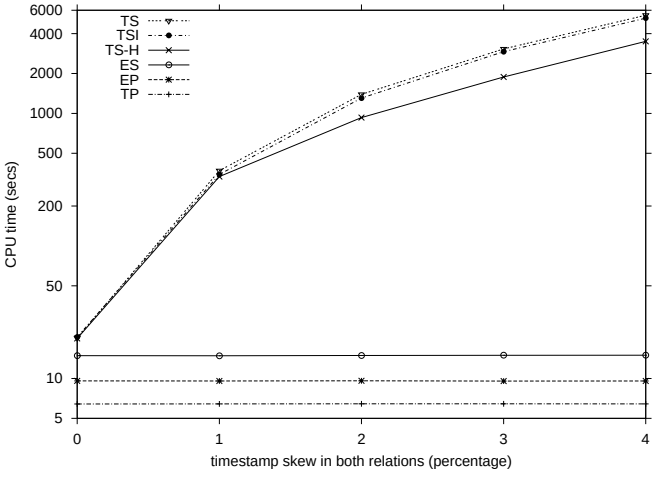


**Fig. 24.** Timestamp attribute skew in both relations

ing algorithms by EP-H, TSI and TSI-H by TSI, and TEP, TP, ETP, EP, ETS-H, and ES-H by EP; the remaining algorithms are retained.

The algorithms fall into three groups. All the timestamp sort-merge algorithms exhibit poor performance. However, the performance of TS and TS-H is much better than that of TSI and TSI-H. At the highest skew, the performance of TS is 174 times better than that of TSI. This is due to the cache overflow in TSI. One percent of 64 MB is 640 KB, which is ten times the cache size. The interval join algorithm scans and purges the cache once for every tuple to be joined. The cache thrashing occurs when the cache overflows. As before, there is no cache overflow in TS and TS-H. The performance gap between these two algorithms and the group with flat curves is caused by in-memory join operations. The CPU time used by each algorithm is plotted separately in Fig. 24. In this figure, all the explicit sort-merge algorithms are represented by ES, all the explicit partitioning algorithms by EP, all the timestamp partitioning algorithms by TP, and TSI and TSI-H by TSI ; the remaining algorithms are retained. Since all but timestamp sort-merge algorithms perform the in-memory join by sorting the relations or the partitions on the explicit attribute, their performance is not at all affected by dual skew.

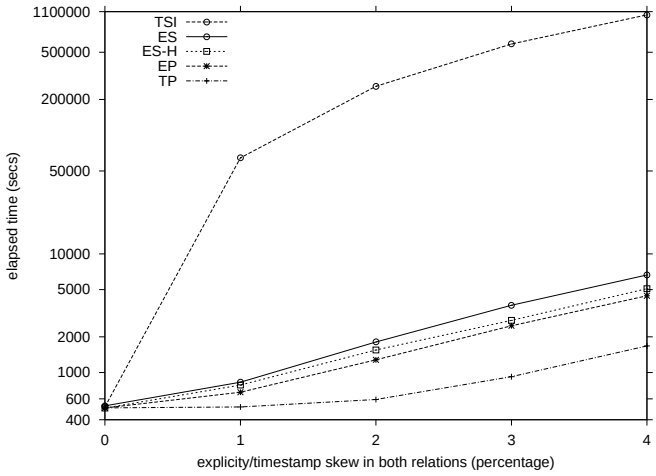


**Fig. 25.** Explicit/timestamp attribute skew in both relations

It is interesting that the CPU time spent by TSI is less than that spent by TS. The poor overall performance of TSI due to the cache overflow can be improved by increasing the cache size of TSI. Actually, TSI performs the join operation in the two caches rather than in the input buffers. Therefore, a large cache size can be chosen when dual skew is present to avoid cache thrashing. In this case, a 1-MB cache size for TSI will result in a performance similar to that of TS.

### *5.9 Explicit/timestamp dual skew*

In this section, we investigate the simultaneous effect of dual skew in both the explicit attribute and the timestamp. This is a challenging situation for any temporal join algorithm.

The memory size is 16 MB, and we generated a series of 64-MB relations with increasing explicit and timestamp chunky skew, from 0% to 4% in 1% increments. Dual skew was guaranteed by performing a self-join on these relations. The results are shown in Fig. 25. In this figure, TSI and TSI-H are represented by TSI, TS and ES by ES, TS-H and ES-H by TS-H, all the explicit partitioning algorithms by EP, and the remaining algorithms by TP.

The interesting point is that all the algorithms are affected by the simultaneous dual skew in both the explicit and timestamp attributes. But they fall into two groups. The algorithms that are sensitive to the dual skew in either explicit attribute or timestamp attribute perform as badly as they do in the experiments described in Sects. 5.7 and 5.8. The performance of the algorithms not affected by the dual skew in either explicit attribute or timestamp attribute degrades with increasing skew. However, their performance is better than that of the algorithms in the first group. This is due to the orthogonality of the explicit skew and the timestamp skew.

### *5.10 Summary*

The performance study described in this section is the first comprehensive, empirical analysis of temporal join algorithms. We investigated the performance of 19 non-index-based join algorithms, namely, nested-loop (NL), explicit

partitioning (EP and EP-H), explicit sorting (ES and ES-H), timestamp sorting (TS and TS-H), interval join (TSI and TSI-H), timestamp partitioning (TP and TP-H), combined explicit/timestamp sorting (ETS and ETS-H) and timestamp/explicit sorting (TES and TES-H), and combined explicit/timestamp partitioning (ETP and ETP-H) and timestamp/explicit partitioning (TEP and TEP-H) for the temporal equijoin. We varied the following main aspects in the experiments: the presence of long-duration timestamps, the relative sizes of the input relations, and the explicit-join and timestamp attribute distributions.

The findings of this empirical analysis can be summarized as follows.

- The algorithms need to be engineered well to avoid performance hits. Care needs to be taken in sorting, in purging the cache, in selecting the next tuple in the merge step, in allocating memory, and in handling intrinsic skew.
- Nested-loop is not competitive.
- The timestamp sorting algorithms, TS, TS-H, TES, TES-H, TSI, and TSI-H, were also not competitive. They were quite sensitive to the duration of input tuple timestamps. TSI and TSI-H had very poor performance in the presence of large amounts of skew due to cache overflow.
- The GRACE variants were competitive only when there was low selectivity and a large memory size relative to the size of the input relations. In all other cases, the hybrid variants performed better.
- In the absence of explicit and timestamp skew, our results parallel those from conventional query evaluation. In particular, when attribute distributions are random, all sorting and partitioning algorithms (other than those already eliminated as noncompetitive) have nearly equivalent performance, irrespective of the particular attribute type used for sorting or partitioning.
- In contrast with previous results in temporal coalescing [5], the binary nature of the valid-time equijoin allows an important optimization for partition-based algorithms. When one input relation is small relative to the available main memory buffer space, the partitioning algorithms have uniformly better performance than their sort-based counterparts.
- The choice of timestamp or explicit partitioning depends on the presence or absence of skew in either attribute dimension. Interestingly, the performance differences are dominated by main memory effects. The timestamp partitioning algorithms were less affected by increasing skew.
- ES and ES-H were sensitive to explicit dual skew.
- The performance of the partition-based algorithms, EP and EP-H, was affected by both outer and dual explicit attribute skew.
- The performance of TP and TP-H degraded when outer skew was present. Except for this one situation, these partition-based algorithms are generally more efficient than their sort-based counterparts since sorting, and associated main memory operations, are avoided.
- It is interesting that the combined explicit/timestamp-based algorithms can mitigate the effect of either explicit attribute skew or timestamp skew. However, when dual skew was present in the explicit attribute and the timestamp simultaneously, the performance of all the algorithms degraded, though again less so for timestamp partitioning.

## 6 Conclusions and research directions

As a prelude to investigating non-index-based temporal join evaluation, this paper initially surveyed previous work, first describing the different temporal join operations proposed in the past and then describing join algorithms proposed in previous work. The paper then developed evaluation strategies for the valid-time equijoin and compared the evaluation strategies in a sequence of empirical performance studies. The specific contributions are as follows.

- We defined a taxonomy of all temporal join operators proposed in previous research. The taxonomy is a natural one in the sense that it classifies the temporal join operators as extensions of conventional operators, irrespective of special joining attributes or other model-specific restrictions. The taxonomy is thus model independent and assigns a name to each temporal operator consistent with its extension of a conventional operator.
- We extended the three main paradigms of query evaluation algorithms to temporal databases, thereby defining the space of possible temporal evaluation algorithms.
- Using the taxonomy of temporal join algorithms, we defined 19 temporal equijoin algorithms, representing the space of all such possible algorithms, and placed all existing work into this framework.
- We defined the space of database parameters that affect the performance of the various join algorithms. This space is characterized by the distribution of the explicit and timestamp attributes in the input relation, the duration of timestamps in the input relations, the amount of main memory available to the join algorithm, the relative sizes of the input relations, and the amount of dual attribute and/or timestamp skew for each of the relations.
- We empirically compared the performance of the algorithms over this parameter space.

Our empirical study showed that some algorithms can be eliminated from further consideration: NL, TS, TS-H, TES, TES-H, ES, ES-H, EP, and EP-H. Hybrid variants generally dominated GRACE variants, eliminating ETP, TEP, and TP. When the relation sizes were different, explicit sorting (ETS, ETS-H, ES, ES-H) performed poorly.

This leaves three algorithms, all partitioning ones: ETP-H, TEP-H, TP-H. Each dominates the other two in certain circumstances, but TP-H performs poorly in the presence of timestamp and attribute skew and is significantly more complicated to implement. Of the other two, ETP-H came out ahead more often than TEP-H. Thus we recommend ETP-H, a hybrid variant of explicit partitioning that partitions primarily by the explicit attribute. If this attribute is skewed so that some buckets do not fit in memory, a further partition on the timestamp attribute increases the possibility that the resulting buckets will fit in the available buffer space.

The salient point of this study is that simple modifications to an existing conventional evaluation algorithm (EP) can be used to effect temporal joins with acceptable performance and at relatively small development cost. While novel algorithms

(such as TP-H) may have better performance in certain circumstances, well-understood technology can be easily adapted and will perform acceptably in many situations. Hence database vendors wishing to implement temporal join may do so with a relatively low development cost and still achieve acceptable performance.

The above conclusion focuses on independent join operations rather than a query consisting of several algebraic operations. Given the correlation between various operations, the latter is more complex. For example, one advantage of sort-merge algorithms is that the output is also sorted, which can be exploited in subsequent operations. This interesting order is used in traditional query optimization to reduce the cost of the whole query. We believe temporal query optimization can also take advantage of this [50]. Among the sort-merge algorithms we have examined, the output of explicit algorithms (ES, ES-H, ETS, ETS-H) is sorted by the explicit join attribute; interval join algorithms produce the output sorted by the start timestamp. Of these six algorithms, we recommend ETS-H due to its higher efficiency.

Several directions for future work exist. Important problems remain to be addressed in temporal query processing, in particular with respect to temporal query optimization. While several researchers have investigated algebraic query optimization, little research has appeared with respect to cost-based temporal query optimization.

In relation to query evaluation, additional investigation of the algorithm space described in Sect. 5 is needed. Many optimizations originally developed for conventional databases, such as read-ahead and write-behind buffering, forecasting, eager and lazy evaluation, and hash filtering, should be applied and investigated. Cache size and input buffer allocation tuning is also an interesting issue.

All of our partitioning algorithms generate maximal partitions, that of the main memory size minus a few blocks for the left-hand relation of the join, and then apply that partitioning to the right-hand relation. In the join step, a full left-hand partition is brought into main memory and joined with successive blocks from the associated right-hand partition. Sitzmann and Stuckey term this a *static buffer allocation* strategy and instead advocate a *dynamic buffer allocation* strategy in which the left-hand and right-hand relations are partitioned in one step, so that two partitions, one from each relation, can simultaneously fit in the main memory buffer [49]. The advantage over the static strategy is that fewer seeks are required to read the right-hand side partition; the disadvantage is that this strategy results in smaller, and thus more numerous, partitions, which increases the number of seeks and requires that the right-hand side also be sampled, which also increases the number of seeks. It might be useful to augment the timestamp partitioning to incorporate dynamic buffer allocation, though it is not clear at the outset that this will yield a performance benefit over our TP-H algorithm or over ETP-H.

Dynamic buffer allocation for conventional joins was first proposed by Harris and Ramamohanarao [22]. They built the cost model for nested loop and hash join algorithms with the size of buffers as one of the parameters. Then for each algorithm they computed the optimal, or suboptimal but still good, buffer allocation that led to the minimum join cost. Finally, the optimal buffer allocation was used to perform the join. It would be interesting to see if this strategy can improve the performance of temporal joins. It would also be useful to develop cost models for the most promising temporal join algorithm(s), starting with ETP-H.

The next logical progression in future work is to extend this work to index-based temporal joins, again investigating the effectiveness of both explicit attribute indexing and timestamp indexing. While a large number of timestamp indexes have been proposed in the literature [44] and there has been some work on temporal joins that use temporal or spatial indexes [13, 33,52,56], a comprehensive empirical comparison of these algorithms is needed.

Orthogonally, more sophisticated techniques for temporal database implementation should be considered. In particular, we expect specialized temporal database architectures to have a significant impact on query processing efficiency. It has been argued in previous work that incremental query evaluation is especially appropriate for temporal databases [24, 34,41]. In this approach, a query result is materialized and stored back into the database if it is anticipated that the same query, or one similar to it, will be issued in the future. Updates to the contributing relations trigger corresponding updates to the stored result. The related topic of global query optimization, which attempts to exploit commonality between multiple queries when formulating a query execution plan, also has yet to be explored in a temporal setting.

*Acknowledgements.* This work was sponsored in part by National Science Foundation Grants IIS-0100436, CDA-9500991, EAE-0080123, IRI-9632569, and IIS-9817798, by the NSF Research Infrastructure Program Grants EIA-0080123 and EIA-9500991, by the Danish National Centre for IT-Research, and by grants from Amazon.com, the Boeing Corporation, and the Nykredit Corporation.

We also thank Wei Li and Joseph Dunn for their help in implementing the temporal join algorithms.

## References

1. Allen JF (1983) Maintaining knowledge about temporal intervals. Commun ACM 26(11):832–843
2. Arge L, Procopiuc O, Ramaswamy S, Suel T, Vitter JS (1998) Scalable sweeping-based spatial join. In: Proceedings of the international conference on very large databases, New York, 24–27 August 1998, pp 570–581
3. Beckmann N, Kriegel HP, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the ACM SIGMOD conference, Atlantic City, NJ, 23–25 May 1990, pp 322–331
4. van den Bercken J, Seeger B (1996) Query processing techniques for multiversion access methods. In: Proceedings of the international conference on very large databases, Mubai (Bombay), India, 3–6 September 1996, pp 168–179
5. Böhlen MH, Snodgrass RT, Soo MD (1997) Temporal coalescing. In: Proceedings of the international conference on very large databases, Athens, Greece, 25–29 August 1997, pp 180–191
6. Clifford J, Croker A (1987) The historical relational data model (HRDM) and algebra based on lifespans. In: Proceedings of the international conference on data engineering, Los Angeles, 3–5 February 1987, pp 528–537. IEEE Press, New York
7. Clifford J, Croker A (1993) The historical relational data model (HRDM) revisited. In: Tansel A, Clifford J, Gadia S, Jajodia S,

Segev A, Snodgrass RT (eds) Temporal databases: theory, design, and implementation, ch 1. Benjamin/Cummings, Reading, MA, pp 6–27
8. Clifford J, Uz Tansel A (1985) On an algebra for historical relational databases: two views. In: Proceedings of the ACM SIGMOD international conference on management of data, Austin, TX, 28–31 May 1985, pp 1–8
9. DeWitt DJ, Katz RH, Olken F, Shapiro LD, Stonebraker MR, Wood D (1984) Implementation techniques for main memory database systems. In: Proceedings of the ACM SIGMOD international conference on management of data, Boston, 18–21 June 1984, pp 1–8
10. Dittrich JP, Seeger B, Taylor DS, Widmayer P (2002) Progressive merge join: a generic and non-blocking sort-based join algorithm. In: Proceedings of the conference on very large databases, Madison, WI, 3–6 June 2002, pp 299–310
11. Dunn J, Davey S, Descour A, Snodgrass RT (2002) Sequenced subset operators: definition and implementation. In: Proceedings of the IEEE international conference on data engineering, San Jose, 26 February–1 March 2002, pp 81–92
12. Dyreson CE, Snodgrass RT (1993) Timestamp semantics and representation. Inform Sys 18(3):143–166
13. Elmasri R, Wuu GTJ, Kim YJ (1990) The time index: an access structure for temporal data. In: Proceedings of the conference on very large databases, Brisbane, Queensland, Australia, 13–16 August 1990, pp 1–12
14. Etzion O, Jajodia S, Sripada S (1998) Temporal databases: research and practice. Lecture notes in computer science, vol 1399. Springer, Berlin Heidelberg New York
15. Gadia SK (1988) A homogeneous relational model and query languages for temporal databases. ACM Trans Database Sys 13(4):418–448
16. Gao D, Jensen CS, Snodgrass RT, Soo MD (2002) Join operations in temporal databases. TimeCenter TR-71 `http://www.cs.auc.dk/TimeCenter/pub.htm`
17. Gao D, Kline N, Soo MD, Dunn J (2002) TimeIT: the Time integrated testbed, v. 2.0 Available via anonymous FTP at: `ftp.cs.arizona.edu`
18. Graefe G (1993) Query evaluation techniques for large databases. ACM Comput Surv 25(2):73–170
19. Graefe G, Linville A, Shapiro LD (1994) Sort vs. hash revisited. IEEE Trans Knowl Data Eng 6(6):934–944
20. Gunadhi H, Segev A (1991) Query processing algorithms for temporal intersection joins. In: Proceedings of the IEEE conference on data engineering, Kobe, Japan, 8–12 April 1991, pp 336–344
21. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: Proceedings of the ACM SIGMOD conference, Boston, 18–21 June 1984, pp 47–57
22. Harris EP, Ramamohanarao K (1996) Join algorithm costs revisited. J Very Large Databases 5(1):64–84
23. Jensen CS (ed) (1998) The consensus glossary of temporal database concepts – February 1998 version. In [14], pp 367–405
24. Jensen CS, Mark L, Roussopoulos N (1991) Incremental implementation model for relational databases with transaction time. IEEE Trans Knowl Data Eng 3(4):461–473
25. Jensen CS, Snodgrass RT, Soo MD (1996) Extending existing dependency theory to temporal databases. IEEE Trans Knowl Data Eng 8(4):563–582
26. Jensen CS, Soo MD, Snodgrass RT (1994) Unifying temporal models via a conceptual model. Inform Sys 19(7):513–547
27. Leung TY, Muntz R (1990) Query processing for temporal databases. In: Proceedings of the IEEE conference on data engineering, Los Angeles, 6–10 February 1990, pp 200–208
28. Leung TYC, Muntz RR (1992) Temporal query processing and optimization in multiprocessor database machines. In: Proceedings of the conference on very large databases, Vancouver, BC, Canada, pp 383–394
29. Leung TYC, Muntz RR (1993) Stream processing: temporal query processing and optimization. In: Tansel A, Clifford J, Gadia S, Jajodia S, Segev A, Snodgrass RT (eds) Temporal databases: theory, design, and implementation, ch 14, Benjamin/Cummings, Reading, MA, pp 329–355
30. Li W, Gao D, Snodgrass RT (2002) Skew handling techniques in sort-merge join. In: Proceedings of the ACM SIGMOD conference on management of data Madison, WI, 3–6 June 2002, pp 169–180
31. Lo ML, Ravishankar CV (1994) Spatial joins using seeded trees. In: Proceedings of the ACM SIGMOD conference, Minneapolis, MN, 24–27 May 1994, pp 209–220
32. Lo ML, Ravishankar CV (1996) Spatial hash-joins. In: Proceedings of ACM SIGMOD conference, Montreal, 4–6 June 1996, pp 247–258
33. Lu H, Ooi BC, Tan KL (1994) On spatially partitioned temporal join. In: Proceedings of the conference on very large databases, Santiago de Chile, Chile, 12–15 September 1994, pp 546–557
34. McKenzie E (1988) An algebraic language for query and update of temporal databases. Ph.D. dissertation, Department of Computer Science, University of North Carolina, Chapel Hill, NC
35. Mishra P, Eich M (1992) Join processing in relational databases. ACM Comput Surv 24(1):63–113
36. Navathe S, Ahmed R (1993) Temporal extensions to the relational model and SQL. In: Tansel A, Clifford J, Gadia S, Jajodia S, Segev A, Snodgrass RT (eds) Temporal databases: theory, design, and implementation. Benjamin/Cummings, Reading, MA, pp 92–109
37. Orenstein JA (1986) Spatial query processing in an object-oriented database system. In: Proceedings of the ACM SIGMOD conference, Washington, DC, 28–30 May 1986, pp 326–336
38. Orenstein JA, Manola FA (1988) PROBE spatial data modeling and query processing in an image database application. IEEE Trans Software Eng 14(5):611–629
39. Özsoyoğlu G, Snodgrass RT (1995) Temporal and real-time databases: a survey. IEEE Trans Knowl Data Eng 7(4):513–532
40. Patel JM, DeWitt DJ (1996) Partition based spatial-merge join. In: Proceedings of the ACM SIGMOD conference, Montreal, 4–6 June 1996, pp 259–270
41. Pfoser D, Jensen CS (1999) Incremental join of time-oriented data. In: Proceedings of the international conference on scientific and statistical database management, Cleveland, OH, 28–30 July 1999, pp 232–243
42. Ramakrishnan R, Gehrke J (2000) Database management systems. McGraw-Hill, New York
43. Rana S, Fotouhi F (1993) Efficient processing of time-joins in temporal data bases. In: Proceedings of the international symposium on DB systems for advanced applications, Daejeon, South Korea, 6–8 April 1993, pp 427–432
44. Salzberg B, Tsotras VJ (1999) Comparison of access methods for time-evolving data. ACM Comput Surv 31(2):158–221
45. Samet H (1990) The design and analysis of spatial data structures. Addison-Wesley, Reading, MA
46. Segev A (1993) Join processing and optimization in temporal relational databases. In: Tansel A, Clifford J, Gadia S, Jajodia S, Segev A, Snodgrass RT (eds) Temporal databases: theory, design, and implementation, ch 15. Benjamin/Cummings, Reading, MA, pp 356–387

47. Segev A, Gunadhi H (1989) Event-join optimization in temporal relational databases. In: Proceedings of the conference on very large databases, Amsterdam, 22–25 August 1989, pp 205–215
48. Sellis T, Roussopoulos N, Faloutsos C (1987) The $R^+$-tree: a dynamic index for multidimensional objects. In: Proceedings of the conference on very large databases, Brighton, UK, 1–4 September 1987, pp 507–518
49. Sitzmann I, Stuckey PJ (2000) Improving temporal joins using histograms. In: Proceedings of the international conference on database and expert systems applications, London/Greenwich, UK, 4–8 September 2000, pp 488–498
50. Slivinskas G, Jensen CS, Snodgrass RT (2001) A foundation for conventional and temporal query optimization addressing duplicates and ordering. Trans Knowl Data Eng 13(1):21–49
51. Snodgrass RT, Ahn I (1986) Temporal databases. IEEE Comput 19(9):35–42
52. Son D, Elmasri R (1996) Efficient temporal join processing using time index. In: Proceedings of the conference on statistical and scientific database management, Stockholm, Sweden, 18–20 June 1996, pp 252–261
53. Soo MD, Jensen CS, Snodgrass RT (1995) An algebra for TSQL2. In: Snodgrass RT (ed) The TSQL2 temporal query language, ch 27, Kluwer, Amsterdam, pp 505–546
54. Soo MD, Snodgrass RT, Jensen CS (1994) Efficient evaluation of the valid-time natural join. In: Proceedings of the international conference on data engineering, Houston, TX, 14–18 February 1994, pp 282–292
55. Tsotras VJ, Kumar A (1996) Temporal database bibliography update. ACM SIGMOD Rec 25(1):41–51
56. Zhang D, Tsotras VJ, Seeger B (2002) Efficient temporal join processing using indices. In: Proceedings of the IEEE international conference on data engineering, San Jose, 26 February–1 March 2002, pp 103–113
57. Zurek T (1997) Optimisation of partitioned temporal joins. Ph.D. Dissertation, Department of Computer Science, Edinburgh University, Edinburgh, UK