Join Operations in Temporal Databases

Georgios Giannikis

Supervisor: Prof. Donald Kossmann
Shepherd: Lucia Ambrošová

Contents

1 Introduction **2**

2 Mathematical Definitions **3**
2.1 Generic Definitions 3
2.2 Temporal Join Operations 4
2.3 Reducibility 5

3 Algorithmic Framework **6**
3.1 Taxonomy 6

4 Engineering the Algorithms **7**

5 Performance Evaluation **9**

6 Conclusion **10**

1 Introduction

This report is based on a paper[2] from Dengfeng Gao et al and summarizes the challenges of performing join operations over temporal relational data. Time is an attribute of all real-world phenomena. Database management systems is one domain that are expected to handle such temporal data, as there exist a number of scenarios where time related information is important.

Joins are arguably the most important relational operators. Literature reports an ever-increasing number of research that focuses on optimizing join operators for particular use cases. This is so because efficient join processing is essential for the overall efficiency of a query processor. Since database normalization is necessary to avoid data replication and increase performance, joins occur frequently in almost all existing database deployments. Poor implementations of join operations are tantamount to computing the Cartesian product of the input relations.

By introducing the time attribute in a database system, the problem evolves further. Existing techniques are aimed at the optimization of joins with equality predicates, rather than the inequality predicates, which is more frequent in temporal queries. Moreover, the introduction of a time dimension may significantly increase the size of the database, since it requires storing the multiple versions of each tuple and the time information. These factors indicate that new techniques are required to efficiently evaluate joins over temporal relations.

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. We consider an important subproblem of temporal query processing, the evaluation of ad hoc temporal join operations, i.e., join operations for which indexing or secondary access paths are not available or appropriate.

The purpose of this report is to present a comprehensive survey of semantics and implementation details of existing join operations in temporal databases. There is a number of temporal join operators that 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. 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. A database may support none, one, or both of the given time dimensions. In this report, 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 not considered in this report, though many of the described techniques extend readily to bitemporal databases. In this report, we will use the term non-temporal to refer to databases that provide no integrated support for time.

2 Mathematical Definitions

Before presenting the join algorithms, we have to define the abstract mathematical expressions for the temporal relational joins. The need for a mathematical definition comes from the data model of these operators that is extended to include temporal information. A proper mathematical definition will allow proper classification and comparison of all defined temporal join algorithms.

2.1 Generic Definitions

The first extension of the data model is to introduce temporal counterparts in the relational schemas. Given two relational schemas $R = (A_1, \ldots, A_n)$ and $S = (B_1, \ldots, B_m)$, we extend them by introducing two temporal attributes, T_s and T_e which are the timestamp start and end attributes respectively. The timestamp attributes define a range, during which the record's attributes hold. The extended schemas are now defined as $R = (A_1, \ldots, A_n, T_s, T_e)$ and $S = (B_1, \ldots, B_m, T_s, T_e)$. The $A_i, 1 \leq i \leq n$ and $B_i, 1 \leq i \leq m$ are called explicit attributes as they define non-temporal data.

We will use T as shorthand for the interval $[T_s, T_e]$ 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. Next, we provide five auxiliary definitions over temporal relational data:

Chronon: A *chronon* is the minimal-duration interval (quantum) in which we partition the timeline.

Overlap: $Overlap(U, V)$ returns the maximum interval contained in its two argument intervals if they exist, or $\emptyset$ if such interval does not exist.

Coalesce: The $coalesce(r)$ function collapses value-equivalent tuples – tuples with mutually equal explicit attribute values – in a temporal relation, into a single tuple, that contains a finite union of time intervals.

Expand: The $expand(U)$, function returns the set of maximal intervals contained in its argument temporal element.

2.2 Temporal Join Operations

We begin by examining the core set of non-temporal relational joins that have long been accepted as "standard": 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.

Temporal Cartesian Product The Temporal Cartesian Product, $r \times^T s$ is defined as follows:

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

The output relation z, contains all the explicit attributes of A and B, and additionally has an interval that is the overlap of the original intervals. The last part of the definition ensures that the time interval of the result is nonempty. Basically, the Temporal Cartesian Product can be interpreted as a non-temporal Cartesian Product applied independently at each point in time. When operating on interval stamped data, these semantics correspond to an intersection: the result will be valid during those times when contributing tuples from both input relations are valid.

Temporal Theta Join The Temporal Theta Join operator, $r \bowtie_P^T s$, is defined as a selection of the those tuples from $r \times^T s$ that satisfy an unrestricted predicate P on the explicit attributes of either r or s. The formal definition is:

$$r \bowtie_P^T s = \sigma_{P(r[A],s[B])}(r \times^T s)$$

Temporal Equijoin The Temporal Equijoin is a subclass of the Theta Join where the predicate is strictly an equality predicate matching explicit attributes of the two relations. The mathematical definition of the Temporal Equijoin is:

$$r \bowtie_{r[A']=s[B']}^T s$$

The Equijoin is a very common operation in not temporal databases. In Temporal Databases, inequality join predicates are more often, mostly on the time interval attributes.

A specialization of the Temporal Equijoin is the Temporal Natural Join, which requires that the two input relations, r and s, have a number of identically named explicit attributes. These attributes are used as the equality predicates of the Temporal Equijoin.

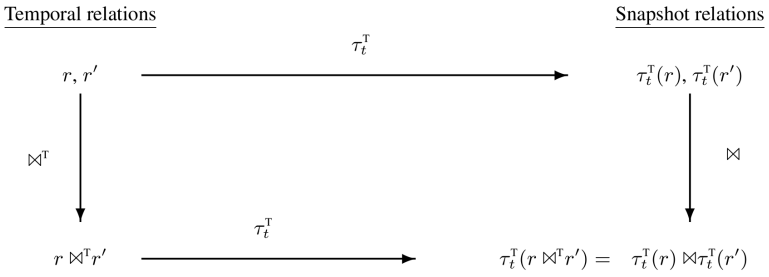

Figure 1: Reducibility of Temporal Natural Join to Snapshot Natural Join

Temporal Outer Joins An outer join does not require each tuple in the two joined relations to have a matching tuple. The joined relation retains each tuple, even if no other matching tuple exists. Outer joins subdivide further into left outer joins, right outer joins, and full outer joins, depending on which relation(s) one retains the rows from (left, right, or both).

In a Temporal Database, all the tuples, including the retained not-matching tuples, must preserve the temporal property of having a common time interval. Mathematically, this is performed using the $coalesce(r)$ and $expand(T)$ helper functions. This way, on a left outer join, $r \ltimes^{T}_{r[A']=s[B']} s$, we have to enforce the following condition to include all the tuples that do not have a match in respect to their explicit attributes, but their intervals overlap:

$$\begin{aligned}&\exists x \in coalesce(r) \forall y \in coalesce(s)\\&(x[A'] \neq y[B'] \Rightarrow z[A] = x[A] \wedge z[B] = null\\&\wedge z[T] \in expand(x[T]) \wedge z[T] \neq \emptyset)\end{aligned}$$

The Temporal Right Outer Join is symmetrical to the Left Outer Join. Finally, the Temporal Full Outer Join can be defined as the union of the Left Outer Join and the Right Outer Join.

2.3 Reducibility

Having defined the join operations on temporal data, we proceed to show how the temporal operations can be reduced to non-temporal operations. This is called Reducibility and guarantees that the semantics of the non-temporal operator is preserved in its more complex temporal counterpart. Reducability is possible by introducing the timeslice operation τ^T, which takes a temporal relation r as argument and a chronon t as parameter and returns the corresponding snapshot relation. The reducibility diagram for Natural Join is shown in Figure 1. Other joins follow the same rules.

3 Algorithmic Framework

Having described the semantics of the previously proposed temporal join operators, we will continue by examining the various implementation details of the state of the art systems. First we need to identify the space of algorithms applicable to temporal join operators. This will allow the creation of a consistent framework within which all existing temporal join algorithms can be placed.

The framework is based on paradigms from existing non-temporal databases. However, the nature of time in a database increases the complexity of the system for two different reasons. First of all, there is a need to compare against complex data types, e.g. intervals requiring inequality predicates, which non-temporal database processors are not optimized to handle. Additionally, a temporal database is usually larger in terms of data size, which is a result of the different versions of the tuples that are stored.

3.1 Taxonomy

All the query evaluation algorithms derive from four basic paradigms: nested-loop, partitioning, sort-merge and index-based. We will not consider index-based algorithms that use auxiliary access paths to the tuples of the relations. Such algorithms have been studied extensively in former literature[1].

Nested-loop join algorithms operate by exhaustively comparing pairs of tuples from the input relations. In other words, the algorithm fetches a tuple from relation r and tries to match it against the whole relation s. Being an I/O intensive operation, various optimizations have been proposed, like the block-nested-loop-join, in which the algorithm fetches a block of tuples from r and tries to match it against s.

Partitioning based join algorithms operate on a divide and conquer paradigm, where tuples are placed into buckets based on their join attribute. Corresponding buckets contain all tuples that could possibly match. Similarly, the sort merge based algorithms divide the input relation in sorted buckets that are located in physical memory. These buckets are sorted and then a merge phase over these buckets produces the result.

For a non-temporal relation, sort-based join algorithms order the input relation on the explicit join attributes. However, on a temporal relation, which includes timestamp information, we can identify four different approaches on ordering tuples: order using the explicit attributes exclusively, order using the timestamp attributes, order primarily on explicit attributes and secondarily on time and finally, order primarily on temporal attributes and secondarily on explicit attributes. By duality, the division step of partition-based algorithms can partition the tuples using any of these options. Hence four choices exist for the dual steps of merging in sort-merge or partitioning in partition-based algorithms.

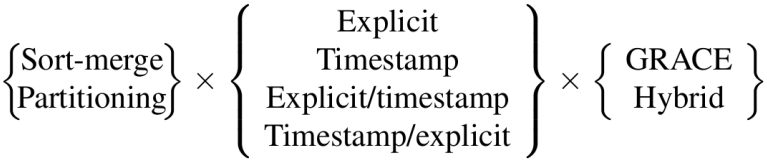

Figure 2: Space of Possible Evaluation Algorithms

As the sort-merge and partition based algorithms require heavy use of buffering, we identify two different buffer allocation strategies that are used in existing algorithms: the GRACE and the hybrid buffer. Both of them dynamically allocate buffer space, however in the hybrid buffer allocation strategy one bucket holding tuples from the outer relation is designated as memory resident. and its buffer space is increased accordingly to hold the whole bucket in memory.

Finally, the new time dimension allows for a new time-based join algorithm, called the interval join. Using knowledge from multi-dimensional spatial joins, the interval join operates similarly to an one-dimensional spatial join over time. Using this transformation, an temporal equijoin can be represented as a two dimensional spatial join, where one dimension is the temporal attribute and the other dimension is the explicit attribute(s). Interval joins, require buffering of tuples, which adds an option for the two different (GRACE, hybrid) buffer allocation strategies.

All these algorithm design choices are summarized in Figure 2. What is omitted is the block-nested loop join algorithm and the interval join with the two different buffer allocation strategies.

Each of these 19 algorithms exhibits different characteristics that make them more appropriate for one problem or another. Next we will implement each of these 19 algorithms in order to evaluate their performance under different scenarios.

4 Engineering the Algorithms

In this section we discuss all the implementation choices that were made while engineering the different algorithms.

Nested Loop Algorithm The Temporal Nested Loop Join algorithm derives from the non-temporal algorithm by including a timestamp predicate at the same time as the predicate on explicit attributes, as illustrated on Figure 3. In order to minimize I/O overhead, we performed joins over blocks of tuples rather than one tuple at a time. The Block-Nested Loop Join is conceptually simple, however the quadratic cost is often non-competitive.

```
For each tuple r in R do
    For each tuple s in S do
       If r and s satisfy the join condition and overlap(r, s)
          Then output the tuple <r, s, overlap(r, s)>
```

Figure 3: Temporal Nested Loop Join

Sort Merge Based Algorithms There are a number of optimizations for sort merge based algorithms. First, we combined the last sort step with the merge step, thereby avoiding one read and one write scan. Then, we used SC-n, a spooled cache on multiple runs technique, which has better performance in the presence of intrinsic skew. In general, sort based algorithms are extremely sensitive to skew, as repetition of attributes values requires rereading of the corresponding tuples. Additionally, in order to introduce the time notion in the algorithm, we created a variant of the algorithm that sorts on timestamps rather than on the join attribute. Timestamp sorting eliminates any skew that may exist over the explicit join attributes, however, it is sensitive to timestamp skew. For completeness, two more variants were constructed by allowing two-way sorting, first over the explicit attribute and then over time and vice-versa. We expect the extra sorting step will not optimize the algorithm but rather simply increase the CPU time.

In order to reduce the I/O operations, we used a specialized purge caching, where cache is purged periodically rather than when it is already full. Another optimization is the use of a Heap for the last run of the sorting phase. This effectively reduces the computational cost from $O(n)$ to $O(log_2 n)$. Finally, we used the two variants of buffer allocation strategies, GRACE and hybrid, that were discussed earlier.

Partition Based Algorithms In partitioning-based algorithms, 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, which increases random I/O. As a compromise, a slightly large outer buffer size that is able to fit 32 pages. A more appropriate choice of the buffers' size is considered future work.

The implemented partition based algorithm fetches the whole outer partition into memory, assuming that it will not overflow the buffer space. The outer partition is then sorted, using an in-memory quicksort algorithm and the remaining memory is used for scanning the inner partition. For each inner tuple, matching outer tuples are found using a 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.

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

Figure 4: Experimental System Characteristics

Finally, as with the sort-merge based algorithms, we implemented two different variants using GRACE and hybrid buffer allocation strategies.

5 Performance Evaluation

The described algorithms were evaluated using a scaled down instance of a database system. The sizes of the different parameters are summarized in Figure 4. We were less interested in absolute system size than in the ratio of data 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.

A number of different experiments was performed in order to identify the advantages and disadvantages of each algorithm. In the first experiment the selectivity of the explicit attributes was very low and the time intervals were very small. This experiment mimics a foreign key-primary key natural join in that the cardinality of the result is the same as one of the input relations. Then the same experiment with longer time intervals was performed. The longer time intervals increase the amount of time-overlapping tuples which should effectively deteriorate the performance of the time-based algorithms. The third experiment was a natural extension of the first two, where a mixture of short and long time intervals was used. The goal of this experiment is to identify the relative overhead of long time intervals.

A second set of experiments was designed to evaluate the performance of these algorithms when the sizes of the two relations vary. For these experiments, both short and long time intervals were used concurrently. This experiment should show that when the input relation is small relative to the available memory, the partition based algorithms will exhibit higher performance as the relation will be stored entirely in main memory.

Then, an evaluation of the performance of all algorithms in case of attribute skew follows. Three types of skew were considered: explicit attribute skew, temporal attribute skew and explicit and temporal attribute skew. The goal of this set of experiments is to identify how sensitive are the algorithms to data skew and on what degree.

After executing all the sets of experiments on all 19 algorithms we are able to understand if a certain algorithm outperforms the others. Of course, in an environment of so many dimensions, it is hard to find a single winner; however, there are many losers.

Furthermore, from the experimental results, it is clear that the nested-loop algorithms are not competitive. The quadratic cost dominates the performance of these algorithms. Also, the timestamp sort-based algorithms are not competitive as they are quite sensitive to the duration of input tuple timestamps. Additionally, these algorithms exhibit very poor performance in the presence of large amounts of skew due to cache overflow. In the absence of explicit and timestamp skew, the results parallel those from non-temporal 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 datasets where skew is present, the choice of timestamp or explicit partitioning depends on which attribute type the skew exists. Interestingly, the performance differences are dominated by main memory effects. In general, timestamp partitioning algorithms were less affected by increasing temporal skew, while explicit partitioning algorithms are heavily affected by explicit attribute skew. Explicit sort-merge based algorithms were non competitive in case of explicit skew, which was expected.

Regarding, buffer allocation strategies, 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.

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 Conclusion

The report surveys a number of existing temporal join algorithms, proposes an algorithmic framework that is used to classify each of the temporal join algorithm and finally, studies the advantages and disadvantages of the different classes of temporal joins. The taxonomy that is used to classify the

algorithms is a natural one, in the sense that it classifies the temporal join operators as extensions of non-temporal operators, irrespective of special joining attributes or other model-specific restrictions.

Based on this taxonomy, a framework of temporal join algorithms is introduced which extends the three main paradigms of query evaluation algorithms to temporal databases, thereby defining the space of possible temporal evaluation algorithms. The framework defines 19 temporal equijoin algorithms, representing the space of all such possible algorithms, and places all existing work into this framework.

Then, the report defines 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.

Finally, experimentation was conducted on this framework of algorithms. The empirical study shows that some algorithms can be eliminated from further consideration due to the poor exhibited performance. Due to the high dimensions of the experiments, it is not clear if there is a single winner between these algorithms. A relatively good compromise that gives good performance on almost all dimensions is an algorithm based on two way partitioning, first on explicit attributes and then on temporal attributes that uses hybrid buffer allocation strategies.

Personal Assessment The paper is well illustrated, based on a real problem. It defines a quite big framework in which all existing work could be classified and based on that framework evaluates the advantages and disadvantages of each algorithm. The authors draw a straight line to separate temporal and non-temporal join algorithms by claiming that all non-temporal systems are heavily optimized for equijoin. However, when it comes to evaluation, they use equijoin to compare the set of algorithms. Additionally, what is missing from the paper is a comparison to a real system and probably experimentation on a bigger scaled system.

References

[1] Salzberg B and Tsotras VJ. Comparison of access methods for time-evolving data. In *ACM Comput Surv 31*, 1999.

[2] Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. Join operations in temporal databases. *VLDB Journal*, 2005.