Sequenced Subset Operators: Definition and Implementation

Joseph Dunn Sean Davey Anne Descour
Richard T. Snodgrass
Department of Computer Science
The University of Arizona, Tucson
{jdunn,sdavey,adescour,rts}@cs.arizona.edu

Abstract

Difference, intersection, semi-join and anti-semi-join may be considered binary subset *operators, in that they all return a subset of their left-hand argument. These operators are useful for implementing SQL's* `EXCEPT`, `INTERSECT`, `NOT IN` *and* `NOT EXISTS`, *distributed queries and referential integrity. Difference-all and intersection-all operate on multi-sets and track the number of duplicates in both argument relations; they are used to implement SQL's* `EXCEPT ALL` *and* `INTERSECT ALL`. *Their* temporally sequenced *analogues, which effectively apply the subset operator at each point in time, are needed for implementing these constructs in temporal databases. These SQL expressions are complex; most necessitate at least a three-way join, with nested* `NOT EXISTS` *clauses. We consider how to implement these operators directly in a DBMS. These operators are interesting in that they can fragment the left-hand validity periods (sequenced difference-all also fragments the right-hand periods) and thus introduce memory complications found neither in their non-temporal counterparts nor in temporal joins and semi-joins. This paper introduces novel algorithms for implementing these operators by ordering the computation so that fragments need not be retained in main memory. We evaluate these algorithms and demonstrate that they are no more expensive than a single conventional join.*

1. Introduction

Due to the temporal nature of nearly all database information, much work is being done to extend non-temporal algorithms to the temporal domain. This paper presents algorithms for four relational operators for which no temporal implementation has been published: sequenced difference, sequenced intersection, sequenced semi-join and sequenced anti-semi-join. These operations are necessary to maintain temporal referential integrity between temporal relations, to implement the `EXCEPT`, `EXCEPT ALL`, `INTERSECT`, `INTERSECT ALL`, `NOT IN` and `NOT EXISTS` in SQL [Melton & Simon 93], and to perform efficient remote querying operations in distributed temporal databases. The challenge in implementing these operations lies primarily in the subtractive nature of anti-semi-join and difference.

Semi-join, anti-semi-join, intersection (and intersection-all) and difference (and difference-all) are similar in that they are all binary operators that return only information from the left-hand relation, so they can be thought of operations that "select a subset of the left-hand side tuples." We thus term these operators "subset operators." In a temporal application, the analogous *sequenced* operators "select a subset of the left-hand side tuples at each point in time, or equivalently, select a subset of the left-hand side tuples over a subset of the times spanned by those tuples."

While in the non-temporal versions of all of these operators the cardinality of the result is less than or equal to that of the left-hand side (LHS) relation, the cardinality of the result of the sequenced analogues is often *greater* than that of the LHS. In fact, in the worst case, the result cardinality is slightly larger than the *product* of the cardinalities of the two underlying relations. This aspect requires great care to ensure that the operation can be performed within the main-memory buffer space since the validity periods of the input relation(s) become fragmented during processing.

In this paper, we first define and differentiate the non-temporal and sequenced variants of difference, intersection, semi-join and anti-semi-join and variants that preserve duplicates: difference-all and intersect-all. We then examine the implementation of each in turn. Sequenced semi-join is straightforward, sequenced anti-semi-join is more difficult, sequenced intersection and sequenced difference are simple given sequenced semi-join and sequenced anti-semi-join, respectively, and sequenced difference-all and sequenced intersect-all are the most challenging to implement. The performance of these algorithms is examined. We show that the performance of our new algorithms is essentially the same as a normal join.

2. Temporal Statements

There are three types of temporal statements (queries, modifications, views, assertions, cursors): current, sequenced and non-sequenced [Snodgrass 00]. Current statements deal with the data using "now" as the time of interest. Non-sequenced statements consider all of the data, without regard to time, i.e. they consider data as it may have been valid "at any time." Sequenced statements deal with data over a specific period or periods and is the most difficult to compute. A sequenced statement is the equivalent of applying the corresponding non-temporal statement at every instant of time and concatenating these results. A statement can be sequenced in valid time, transaction time, or both; in this paper, we restrict ourselves to mono-temporal sequencing (valid or transaction time). The implementation of the sequenced statement should manipulate the underlying periods directly, to efficiently compute the resulting periods.

Consider two relations, `Cow` with attribute `tag` and `Pen` with attributes `tag` and `pen`. The `Cow` relation gives a cow's tenure within the feedlot. The `Pen` relation states when a given cow resided in a particular pen on the feedlot. A particular cow is identified by its `tag`, which is an integer. These relations are time-varying: associated with each is a timestamp indicating the period when the tuple was valid.

We can ask, for which times is a cow's pen known, for all cows in the feedlot. This may be expressed in the following query in the proposed temporal part of SQL3 [Snodgrass et al. 96], as follows (the initial keyword indicates that the statement is sequenced in valid time).

```
VALIDTIME SELECT Cow.tag
FROM Cow, Pen
WHERE Cow.tag = Pen.tag
```

The non-temporal query (without the `VALIDTIME`) is implemented with a semi-join operator; the sequenced variant requires a sequenced semi-join operator.

Consider the referential integrity constraint that `Pen.tag` references `Cow.tag`.

```
ALTER TABLE Pen ADD CONSTRAINT
    FOREIGN KEY (tag) REFERENCES Cow
```

This non-temporal constraint may be implemented with a `NOT EXISTS` subquery.

```
ALTER TABLE Pen ADD CONSTRAINT
NOT EXISTS (SELECT * FROM Pen
 WHERE NOT EXISTS (SELECT * FROM Cow
             WHERE Pen.tag = Cow.tag))
```

The `NOT EXISTS` here can be implemented as anti-semi-join or difference operator.

A sequenced version of this referential constraint might be expressed as follows [Snodgrass et al. 96].

```
ALTER TABLE Pen ADD CONSTRAINT
    FOREIGN KEY (tag) VALIDTIME
        REFERENCES Cow
```

This states that, at each point in time, `Pen.tag` references `Cow.tag`. This sequenced constraint can similarly be implemented with a sequenced anti-semi-join or sequenced difference operator.

3. Operator Definition

In this paper, we will use the following notation: $L \ltimes^T R$ for sequenced semi-join, $L \overline{\ltimes}^T R$ for sequenced anti-semi-join, $L -^T R$ for sequenced difference and $L \cap^T R$ for sequenced intersection. The -all variants (SQL's `EXCEPT ALL` and `INTERSECT ALL`) are denoted by a subscript: $L -^T_{ALL} R$ and $L \cap^T_{ALL} R$. We considered only join predicates that are equalities on the identically-named attributes; the discussion can easily be generalized to equi-join predicates on dissimilarly-named attributes. We term the join attribute-equivalent tuples, *matching tuples*.

3.1. Common Characteristics

Semi-join and intersection are commutative. Anti-semi-join and difference are subtractive operations; they are not commutative. Most join implementations swap the smaller relation into the inner position for improved memory page utilization. Anti-semi-join and difference disallow the swapping of inner and outer relations.

For clarity, we offer examples of each operation using the two relations given below. We can model that the cow with tag 78453 was in the feedlot from January 1994 through April 1997 (here we use a granularity [Bettini et al. 98] of month) and that that cow was in pen 1 from January 1994 through December 1996. We use closed-open periods at a granularity of month as timestamps; these contain the starting month but not the ending month. Hence, the last tuple of `Cow` is only valid during the entire year of 1995.

`Cow`:

`tag`	ValidTime
78453	[1/94–5/97)
12413	[1/95–8/97)
78453	[5/98–∞)
97234	[1/95–1/96)

`Pen`:

`tag`	`pen`	ValidTime
78453	1	[1/94–1/97)
78453	3	[1/97–5/97)
12413	1	[1/95–7/97)
78453	1	[3/98–9/98)
78453	2	[9/98–∞)

The `tag` and `pen` attributes are termed *explicit attributes*, to differentiate them from the ValidTime timestamp. Two tuples in the same table that are identical in the values of their explicit attributes are termed *value-equivalent* [Snodgrass 87].

3.2. Sequenced Semi-Join

A semi-join operation performs the equivalent action of a regular join but only returns information from the LHS relation: $Cow \ltimes Pen = \pi_{Cow}(Cow \bowtie Pen)$. The result contains those tuples that would have participated in a natural (or equi-) join. This characteristic of semi-join operations makes them useful in distributed relational databases, where a smaller result means less time spent in data transfer.

Ignoring the time attribute, $Cow \ltimes Pen$ would return the tags 78453 and 12413. Tag 97234 would be eliminated, because it doesn't participate in the join, by virtue of not residing in the right-hand side (RHS).

We contrast this output with that of an analogous *sequenced semi-join*. The result of $Cow \ltimes^T Pen$ gives the periods for which a cow's pen is known.

$Cow \ltimes^T Pen$:

`tag`	ValidTime
78453	[1/94–1/97)
78453	[1/97–5/97)
12413	[1/95–7/97)
78453	[5/98–9/98)
78453	[9/98–∞)

The first thing we notice is that tag 97234 is again absent, as expected. The second thing we notice is that while the LHS contains four tuples, the result of the semi-join contains five tuples. Compare the above result with the following equivalent sequenced natural join.

$Cow \bowtie^T Pen$:

`tag`	`pen`	ValidTime
78453	1	[1/94–1/97)
78453	3	[1/97–5/97)
12413	1	[1/95–7/97)
78453	1	[5/98–9/98)
78453	2	[9/98–∞)

Only the information stored in `Cow` (the left relation) is returned by a semi-join unlike to the results of a standard equijoin. We now see where the five tuples came from. In this case, the validity period of the first LHS tuple, [1/94–5/97), is broken into two sub-periods, [1/94–1/97) and [1/97–5/97). The same thing happens to the third LHS tuple, because the original period is not found in the corresponding RHS tuple.

3.3. Sequenced Anti-Semi-Join

The anti-semi-join operator returns those LHS tuples that do *not* match the RHS [Graefe 93]: $L \overline{\ltimes} R = L - (L \ltimes R)$. One very important application of the anti-semi-join is maintaining referential integrity. If a referential integrity constraint was specified from `Pen` to `Cow`, the anti-semi-join identifies those tuples in `Pen` that violate this constraint.

A *sequenced anti-semi-join* returns the time complement of a sequenced semi-join. The sequenced anti-semi-join's usefulness lies in its ability to efficiently confirm temporal referential integrity between two relations. (Again, note the `VALIDTIME` keyword.)

```
ALTER TABLE Pen ADD CONSTRAINT
    FOREIGN KEY (tag)
        VALIDTIME REFERENCES Cow
```

We wish to identify tuples (and periods) in `Pen` that violate this constraint. The expression provides the periods for which a pen is specified for a non-existent cow .

$\pi_{tag}(Pen) \overline{\ltimes}^T Cow$:

`tag`	ValidTime
78453	[3/98–5/98)

Going back to the original relations, we see that there indeed a tuple for these two months in `Pen` but not in `Cow`.

In the worst case, all the LHS and RHS tuples have the same value for the join attribute and the LHS tuples are all *long-lived*, that is, their validity period overlaps the validity period of all RHS tuples. Let the cardinalities of the LHS and the RHS relations be l and r, respectively. Each LHS tuple may be split into as many as $r + 1$ short segments, yielding a worst-case cardinality of the result of $l \cdot r + l$.

3.4. Sequenced Difference

There are two variations of the difference operation: difference and difference-all. These two operations are analogous to the SQL constructs `EXCEPT` and `EXCEPT ALL`, respectively. The distinction between the regular difference operation and the difference-all lies in how the RHS matching tuples are counted.

Difference is a specialized form of anti-semi-join, one which requires that the operands be union-compatible. For union-compatible L and R relations, the following holds.

$$\begin{aligned} L \overline{\ltimes} R &= L - (L \ltimes R) \\ &= L - (L \cap R) = L - R \end{aligned}$$

This tautology also holds for sequenced difference, so $L -^T R = L \overline{\ltimes}^T R$. Due to this tautology, a separate implementation of sequenced difference is unnecessary.

3.5. Sequenced Difference-All

Difference-all is distinct from difference and anti-semi-join in an important way: the cardinality of duplicates in both operands must be tracked. Like difference, difference-all subtracts the RHS tuples that match the LHS tuples. But it operates on multi-sets, rather than sets. If a particular tuple appears five times in the LHS and three times in the RHS, the result will contain two copies of this tuple.

These same distinctions apply to sequenced difference and difference-all, but in terms of the validity periods. Both subtract the time periods in the right side matching tuples from the left side matching tuples and then output tuples that contain time periods corresponding to the time remaining in the left side tuples. In sequenced difference, the validity periods for output tuples correspond to those periods that are not overlapped by the times of *any* of the right side matching tuples. For sequenced difference-all, each intersecting period from a particular matching right side tuple is removed from *only one* matching left side tuple.

As an example, the expression $\sigma^T_{pen=1}(\pi^T_{pen}(Pen))$ yields the following L relation.

L:

pen	ValidTime
1	[1/94–1/97)
1	[1/95–7/97)
1	[3/98–9/98)

Notice that duplicates exist at [1/95–1/97). If we sequenced difference-all the above relation with the following R relation,

R:

pen	ValidTime
1	[6/95–6/96)
1	[1/96–5/98)

the following relation results.

$L -^T_{ALL} R$:

pen	ValidTime
1	[1/94–6/95)
1	[6/96–1/97)
1	[1/95–1/96)
1	[5/98–9/98)

Note that the duplicates hold only for [1/95–6/95); in particular, there is only one tuple for [6/95–1/97), because the other was removed by the difference-all. As one might imagine, such period calculations can get complex. Note also that the cardinality of the result is greater than that of the LHS (though, at any point in time, e.g., 2/95, the cardinality of the result, here two tuples during 2/95, is never more than the cardinality of the LHS, here also two during 2/95).

Figure 1 illustrates sequenced difference-all. Here L contains two tuples, as does R. In general, a result tuple starts when a tuple from L starts, or when a tuple from R ends. Analogously, a result tuple ends when a tuple from L ends, or when a tuple from R starts. This gives rise to four cases, as illustrated by the four result tuples shown in the figure. Note that the second tuple in R cancels for its validity period a portion of L, but does not impact the second tuple in L, which it completely overlaps, due to the -all semantics.

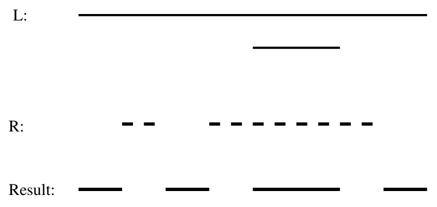

Figure 1. Sequenced difference-all example.

The size of the output relation may still indicate significant processing. As with sequenced difference, sequenced difference-all has a worst-case complexity of $l \cdot r + l$.

3.6. Sequenced Intersection and Intersection-All

As with difference, there are two variations of the intersection operation: intersection and intersection-all, analogous to the SQL constructs `INTERSECT` and

INTERSECT ALL. Intersection is a specialized form of semi-join, one that requires that the operands be union-compatible. This also holds for sequenced intersection: $L \cap^T R = L \ltimes^T R$, Hence, a separate implementation of sequenced intersection is not necessary.

Intersection-all tracks the number of duplicates. If a particular tuple appears five times in the LHS and three times in the RHS, the result will contain three copies of this tuple. Another way of viewing this is that intersection-all removes the extra tuples from the LHS: $L \cap_{ALL} R = L -_{ALL} (L -_{ALL} R)$. This tautology also holds for sequenced intersection-all (tautologies on conventional relational operators always apply to their sequenced analogues) and thus intersection-all can be implemented with difference-all.

4. Related Work

The work that relates to the topic of this paper comes from three areas: (1) implementation of the non-temporal semi-join, difference and anti-semi-join operators, (2) expression of sequenced versions of these operators in SQL [Snodgrass 00] and (3) temporal algorithms for standard joins [Soo et al. 94, Zurek 96].

Semi-join, difference and anti-semi-join receive scant mention in the literature. Graefe classifies as the so-called "one-to-one match operations" all the various joins [Mishra & Eich 92], left and right semi-joins, left, right and symmetric outer-joins, left and right anti-semi-joins, symmetric anti-join, intersection, union, left and right differences and symmetric or anti-difference [Graefe 93]. He goes on to state that "Since all these operations require basically the same steps and can be implemented with the same algorithms, it is logical to implement them in one general and efficient module." [Graefe 93, pp. 104]. While this statement holds for these non-temporal operators, implementation of their temporal analogues is more difficult, due to period fragmentation.

A good summary of proposed temporal algorithms is given by Zurek [Zurek 96]. The majority of previous work in temporal join evaluation has concentrated on refinements of the nested-loop [Segev & Gunadhi 89, Gunadhi & Segev 90], sort-merge algorithms [Leung & Muntz 90] and partition-based algorithms [Leung & Muntz 92, Soo et al. 94].

5. Approach

We now present implementations of the sequenced semi-join, anti-semi-join and difference-all operations. Because these operators are quite similar, many design decisions apply to the implementations of all three. We describe common characteristics before continuing to the individual implementations.

Sequenced semi-join, sequenced anti-semi-join and sequenced difference are similar to standard temporal joins. Therefore, we were able to base our implementations for each of these operations on existing temporal join algorithms. We examined the well-known nested-loop, sort-merge and hash join methods [Mishra & Eich 92]. These techniques are used to select matching tuples; our algorithms come into play in manipulating those tuples and in deciding which modified tuples are to be added to the output relation.

Nested-loop joins are not popular because of their poor performance. The additional memory requirements in the operations we are addressing arise from the existence of temporal elements in the sequenced anti-semi-join and sequenced difference operations. (A *temporal element* is a set of non-overlapping time periods that encapsulates all the time information for a particular fact [Jensen et al. 92].) Temporal elements considerably complicate memory handling as we shall see shortly. A nested-loop-based implementation would need to maintain a number of temporal elements proportional to the number of tuples in the relations, which is clearly not feasible. Therefore we did not consider the nested-loop approach as a basis for our implementations. Instead, we focus on sort-merge join approaches, and touch briefly on hash joins in Section 5.4.

5.1. Sequenced Semi-Join Implementation

Implementing sequenced semi-join is a trivial modification to any existing join algorithm. We include the semi-join operation in order to contrast the simplicity of this implementation with the challenges introduced by the subtractive operations of anti-semi-join and difference-all.

The semi-join property that we will contrast with anti-semi-join is the concept of *tuple-comparison independence*. When left and right join attribute fields match, we compare their respective timestamps and produce an output tuple corresponding to the temporal overlap. This timestamp is completely unaffected by the previously produced tuples; nor will it influence the result of future comparisons. This tuple-comparison independence property does not immediately carry to the anti-semi-join operation or the difference-all operation. Tracking the influence of individual comparisons is what provides added challenge to the subtractive operations.

5.2. Sequenced Anti-Semi-Join Implementation

While sequenced anti-semi-join can be implemented via a modification to existing temporal join algorithms, the changes are not as straight-forward as those required for sequenced semi-join. Figure 2 illustrates this operator applied to a matched tuple set from relations L and R. This figure also shows the final result relation.

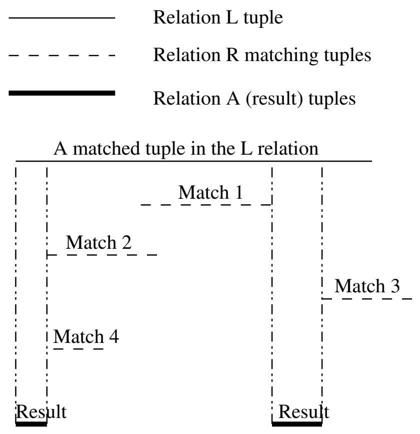
Relation L tuple
Relation R matching tuples
Relation A (result) tuples
A matched tuple in the L relation
Match 1
Match 2
Match 3
Match 4
Result
Result

Figure 2. Sequenced anti-semi-join example demonstrating result tuples.

From Figure 3 we see that a tuple produced by a particular comparison (Match 1) may contain time periods that are not included in the correct result. Both previous and future attribute-equivalent comparisons may remove parts of this tuple's time contribution. Hence, any given tuple's contribution to the output relation is dependent on all other tuples with the same join-attribute value.

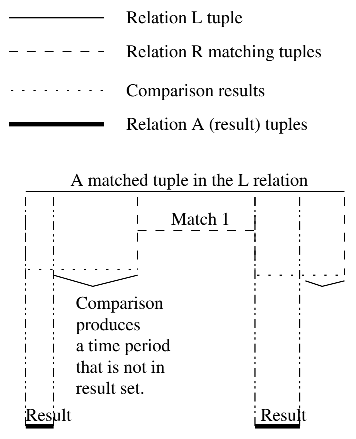
Relation L tuple
Relation R matching tuples
Comparison results
Relation A (result) tuples
A matched tuple in the L relation
Match 1
Comparison produces a time period that is not in result set.
Result
Result

Figure 3. First comparison of our match set.

We considered several possible solutions to this problem. These solutions fell into three general categories: 1) ignore dependence during overlap comparisons, thereby increasing the cost of post-processing; 2) handle dependence by modifying the left tuple's timestamp at the cost of implementing temporal elements; or 3) eliminate the dependence of a tuple's contribution on future comparisons (although past dependence remains) at the cost of pre-processing. Note that we did investigate the first two possibilities, but the details are not presented here due to limited space.

5.2.1 Pre-processing Approach

We can remove the need for post-processing or dynamic memory allocation by exploiting a simple property of time: If RHS tuple r splits LHS tuple l (new periods will be denoted as l_1 and l_2) and the RHS relation is sorted primarily by the join attribute(s), then secondarily by start-time, it is always the case that l_1 cannot be split further. Figure 4 demonstrates this property. The first RHS tuple causes one output tuple to be generated. The second RHS tuple moves the start time of the remaining period to later. The third RHS tuple generates a second output tuple and uses up the remaining timestamp from the LHS tuple.

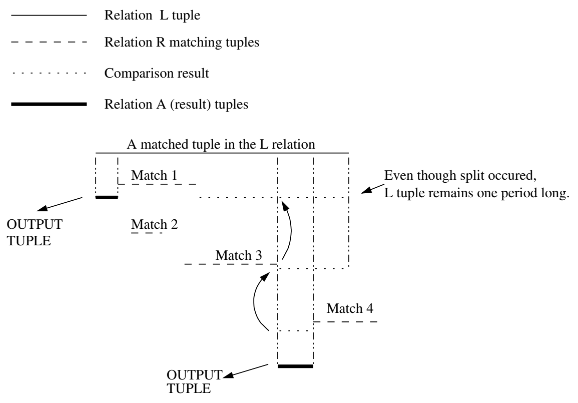
Relation L tuple
Relation R matching tuples
Comparison result
Relation A (result) tuples
A matched tuple in the L relation
Match 1
Even though split occured, L tuple remains one period long.
OUTPUT TUPLE
Match 2
Match 3
Match 4
OUTPUT TUPLE

Figure 4. Handling evaluation dependence via start-time sorting.

By capitalizing on this sorted overlap analysis, we avoid the need for a temporal element on a split-producing overlap, because we need retain only a single period for future comparisons. This single period is easily retained by overwriting the original starting timestamp of l. The pseudo-code for this algorithm, again assuming closed-open period definitions, is given in Figure 5. We see that sorting the RHS on start time eliminates all unnecessary comparisons as well as removing the need for extra memory or for result-relation processing. The algorithm can judge if it is possible for future RHS tuples to modify the LHS tuples; if start-time evaluation indicates no change is possible, inner loop processing is halted.

```
match($l$, $r$):
        result.start ← $l$.start;
        result.stop ← first($l$.stop, $r$.start);
        int x ← last($l$.start, $r$.stop);
        emit($l$.explicit, result.start, result.stop);
        $l$.start ← x;
```

Figure 5. Processing periods in sequenced anti-semi-join.

We stress that time-attribute sorting is only required on the RHS, not on the LHS. (However, both the LHS and the RHS need to be sorted primarily on the join attributes.) As with conventional sort-merge join, all of the matching RHS tuples are (anti-)joined with each LHS tuple independently. If the LHS is sorted, the matching RHS tuples that need to be considered can be reduced (halting comparisons when l's endpoint is less than or equal to r's start point) [Soo et al. 94]. If the RHS is small, the LHS is very large, and the result is expected to be small, then nested-loop may be more attractive. However, if the RHS is small then the cost to sort it is proportionally small—hence, this pre-processing algorithm may still be preferred.

5.3. Sequenced Difference and Difference-All Implementation

Sequenced difference is equivalent to sequenced anti-semi-join when the relations are union compatible. Therefore we won't discuss further the implementation of sequenced difference. (Since sequenced intersection is equivalent to sequenced semi-join and sequenced intersection-all can be expressed in terms of sequenced difference-all we do not discuss implementation of either of these operators.)

Sequenced difference-all must keep track of the numbers of tuples from both sides during each instance of time. Figure 6 shows a graphical representation of the sequenced difference-all operation. Here, two LHS tuples match four RHS tuples on the explicit attributes. The various overlapping tuples partition the time-line into ten segments, each of which must have the correct number of tuples in the result. One tuple results in the first, sixth, eighth, and tenth segments, and two in the second and seventh segments.

Because of the problem of having to maintain a potentially large temporal element, our goal was to find an algorithm that doesn't require additional memory. The

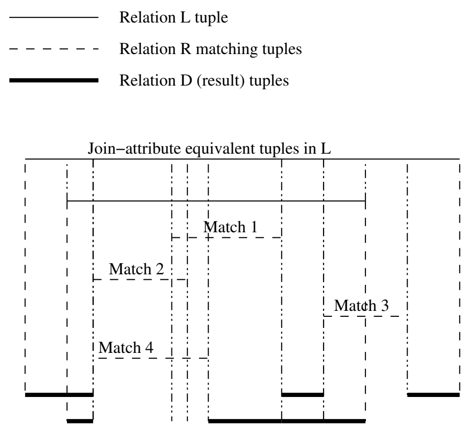
Relation L tuple
Relation R matching tuples
Relation D (result) tuples
Join−attribute equivalent tuples in L
Match 1
Match 2
Match 3
Match 4

Figure 6. Sequenced difference-all.

technique we used for anti-semi-join, outputting the earlier period whenever a split occurs in a temporal element on the left side, can't be used here, because we must also maintain a temporal element for each of the tuples on the right side, as both the LHS and RHS tuples can be fragmented. Initially we investigated a temporal element approach that did not require additional memory. However, we determined that this approach was overly complicated and instead adapted the anti-semi-join preprocessing technique.

We observed in Section 5.2 that sorting the RHS by attribute and sub-sorting the value packets by start time was useful for the anti-semi-join implementation. Expanding on this idea, we investigated sorting both sides in an attempt to compensate for the fact that tuples on the left and right could split into multiple pieces during the course of the algorithm.

Sorting both sides in the same way does not solve the problem, but sorting both by attribute and then subsorting the RHS tuples by *increasing start time* and the LHS by *decreasing stop time* does the job. Due to the sorting of the RHS tuples we achieve the same benefits as in the anti-semi-join. Thus, whenever a LHS tuple is split into two segments the one with the earlier time stamps can be added to the output run immediately since the RHS sorting ensures that no other tuple can alter it. The sorting of the LHS tuples by stop time means that whenever a RHS tuple is split, the resulting tuple with the latest timestamps can be discarded since no other LHS tuple can affect it.

Figure 7 demonstrates the difference-all computation performed on properly sorted left and right hand relations. The first LHS tuple splits the first RHS tuple into two pieces, r_1 and r_2. Due to the sorting of the LHS relation the later resulting piece, r_2, can be discarded. Since the start-time of the first RHS tuple did not change the RHS is still sorted by ascending start-time (see R'). Similarly, when r_1 splits the second LHS tuple into two segments, l_1 and l_2, l_1 can be immediately written to the output run since no other RHS tuple can possibly affect it. Notice that the sorting of the LHS (see L") is preserved since the stop time of the second LHS tuple has not changed. In fact, the only way that the end-time of a LHS tuple l can change is if a RHS tuple r overlaps the end of l. Thus l would be trimmed to the remaining segment, but this could be written to the output run since r has the earliest start-time of any RHS tuple. Similarly, the remainder of r can be discarded since l has the latest end-time of any LHS tuple. Thus the left and right hand sortings are preserved in all cases and so can be relied on as an invariant.

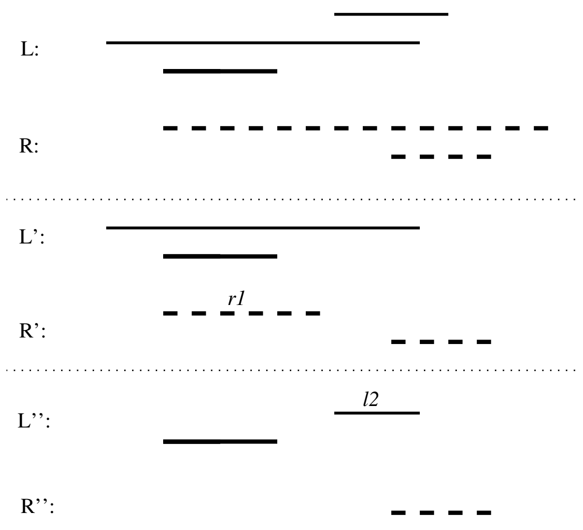

Figure 7. Handling splitting correctly in sequenced difference-all

Figure 8 shows how the timestamps of a pair of tuples, l and r, that match on all the explicit attributes, are manipulated. Interestingly, the only change from Figure 5, for sequenced anti-semi-join, is the fourth line here, in which the stop time of the RHS tuple is altered.

```
match(l, r):
        result.start ← l.start;
        result.stop ← first(l.stop, r.start);
        int x ← last(l.start, r.stop);
        r.stop ← first(l.start, r.stop);
        emit(l.explicit, result.start, result.stop);
        l.start ← x;
```

Figure 8. Processing periods in sequenced difference-all.

5.4. Summary

These approaches are also amenable to hash-based joins, with some care. The LHS and RHS are each hashed according to the join attribute(s). Then a new phase reads each of the RHS partitions and sorts it appropriately. The join phase then reads each LHS partition fully, sorts it appropriately, then merges it block by block with the corresponding RHS partition. Thus the duality that Graefe, Linville and Shapiro previously observed with conventional equi-join [Graefe et al. 94] also applies to subset operators. A critical change is that an extra pass is required over the RHS to sort each partition. This extra pass will probably render hash-based sequenced subset operators less efficient than their sort-based equivalents.

We complete this discussion with two related topics, sort order and temporal coalescing.

The sequenced semi-join operator requires sorting only by the join attribute. The result retains that sort order. The sequenced anti-semi-join requires sorting the LHS on the join attribute; the result will also retain that order. Finally, the sequenced difference-all requires the LHS to be sorted by attribute and then by descending stop time. The result will be sorted by attribute, but not necessarily by stop time.

Temporal coalescing [Böhlen et al. 96, Snodgrass 00] merges value-equivalent tuples whose validity periods meet or overlap, and is similar to duplicate elimination. Coalescing is not applicable to sequenced difference-all nor intersection-all, because duplicates impact the correctness of these operators. However, coalescing of either or both sides will not impact the correctness of sequenced semi-join, anti-semi-join, intersection, nor difference, and can significantly reduce their execution cost. (This is a temporal analogue of projecting out the join attributes during the sorting of the RHS.) The result will be coalesced if the LHS is already coalesced, except for sequenced semi-join, in which the result may not be coalesced even if both the LHS and RHS are coalesced (see Section 3.2).

6. Algorithm Performance

We implemented sequenced semi-join, sequenced anti-semi-join and sequenced difference-all. In previous work [Li et al. 01], we showed that skew in sort-merge join needs to be handled carefully, for both correctness and efficiency. Since the operators in the present paper perform equality matching only on the join attributes, skew is definitely present here. We based our sequenced subset implementations on "spooled cache with multiple runs" (SC-n), a variant of sort-merge join that utilizes a small (32KB) cache to avoid excessive rereading when tuples associated with a particular join attribute value are present both in main memory and on disk. (We note in passing that this is not an issue with the modified hash-join approach mentioned in Section 5.4.) In our previous work, SC-n was shown to exhibit excellent performance, while being much simpler than some of the alternative approaches.

The sorting of the RHS complicates the operation of the tuple cache. Normally tuples could be put into the cache in any order, as long as each LHS tuple is joined with each applicable RHS tuple at some point. Since the sort order is critical to the correctness of the subset join computation, this order must be preserved. When a new LHS tuple is considered, it must be joined first with the cache and then with the RHS tuples in the runs. Furthermore, when tuples are be put into the cache, all the right hand runs must be rewound and added to the cache so that later tuples which will need to join with the cache can join with all the applicable tuples in order.

We implemented both the Grace and the Hybrid [Mishra & Eich 92] variants of each of these algorithms, for a total of six algorithms. We did not apply the optimization of projecting out the join attributes and the ValidTime attribute of the RHS for sequenced semi-join and anti-semi-join, which of course would reduce the I/O cost in writing out the intermediate sorted relation and subsequently reading it back in.

The experiments were performed using the TIMEIT system, which is a software package [Kline & Soo 98] developed to support the prototyping of temporal database components. Some parameters are fixed for all the experiments. They are shown in Table 1(a). In all test cases, the generated relations were randomly ordered.

Parameter	Value
memory size	1MB
cache size	32KB
cluster size	32KB
page size	1KB
tuple size	16 bytes
join attribute	4 bytes

Metric	Conversion
sequential I/O cost	1 msec
random I/O cost	10 msec
attribute compare	20 nsec
attribute move	20 nsec

Table 1. System characteristics and cost metrics.

TIMEIT collects a variety of metrics, shown in Table 1(b); both main memory operations and disk I/O operations were measured. TIMEIT then combines these into a single metric of elapsed time in seconds using the stated weights, thereby not tying the measurements to the underlying processor. We emphasize that this is a computed metric, not actual wall clock time, and so does not capture all of the subtle differences of the algorithms. However, such an approach allows us to understand exactly how each of these metrics is affected by the parameters and algorithms.

Throughout the course of these experiments we fixed several parameters of the sample databases. Each tuple consisted of 16 bytes of which eight were used by the two integer timestamps. The other eight bytes consisted of two integer-sized explicit attributes. Only the first explicit attribute was used as the joining attribute. In all of these operations, the arity of the resulting tuples is equal to that of the tuples in the left-hand relation.

We tested the performance of these operations in a series of experiments that varied the main memory size, tuple fragmentation, LHS cardinality and RHS cardinality. Many sample databases were generated to test the accuracy of the algorithms and our understanding of their behavior. In the interest of space, these results are not shown, as the significant behavior can be characterized by the following analysis.

We used two LHS relations and two RHS relations, carefully designed to ensure identically-sized resulting relations, so that the I/O costs of writing out the result are the same for all six algorithms. Both LHS relations require 1 megabyte, with 65536 tuples. Both RHS relations require 16 MB and contain 1M tuples. The result is always 16MB, with 1M tuples. The tuple order of all input relations was randomized before the subset operator was applied.

The first LHS-RHS relation pair was used with semi-join and anti-semi-join. The LHS has unique join attribute values and each tuple has a lifespan of 950 chronons. The RHS has 16 tuples for each join attribute value, generated by taking the LHS 950-chronon timespans and creating a series of evenly spaced and sized tuples within each timespan (as shown in Figure 9, with three rather sixteen RHS tuples shown). Thus each tuple from the 1MB relation would be split into 16 parts.

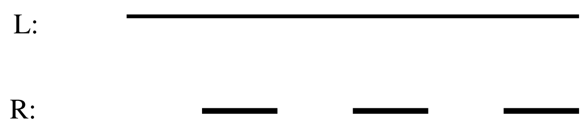

Figure 9. LHS and RHS relations used in testing semi-join and anti-semi-join.

The second LHS-RHS relation pair was used with difference-all. The LHS is in groups of eight tuples with the same join attribute value (to test handling of duplicates), all with a lifespan of 996 chronons. This implies 8,192 unique join attribute values.) The RHS has 16×8 tuples for each join attribute value. The first set of 16 are in the position shown in the previous figure, with subsequent sets of 16 offset by 4 chronons (see Figure 10, again, with just three tuples shown). The difference-all has to contend with the many fragments generated; the result will end up with 16×8 short fragments of various sizes, ranging from four to 32 chronons in length.

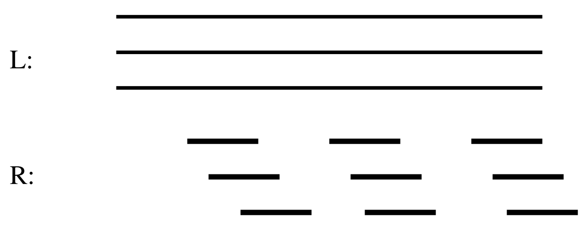

Figure 10. LHS and RHS relations used in testing difference-all.

Figure 11 shows the total computation time (I/O and CPU) for the six algorithms, varying the size of main memory from 0.5MB to 16MB (recall that the LHS relation is 1MB, the RHS relation is 16MB and the resulting relation is 16MB). Note that the y-axis starts at 65 seconds; both axes are linear. This graph shows a clear correlation between the increasing main memory and decreasing computation time. In smaller memory configurations the data must be read and sorted in smaller, more numerous, segments. Reading and writing these smaller segments requires more random I/O than reading and writing larger, more continuous runs. The cost of disk seeks accounts for most of this difference, as shown in Table 2. This table gives, for the smallest and largest memory configurations, the number of seeks, the contribution to the time of those seeks and the time of the rest of the processing (I/O and CPU). After factoring out the number of seeks, there is much less difference in the performance of these algorithms between these two memory configurations.

Figure 12 is a graph of the post-sorting non-I/O costs at various memory sizes (note that here the y-axis starts at 0 seconds). At first it would seem that this graph should be nearly flat because the computation involves the same data regardless of the size of main memory. However, there is a large discrepancy between sequenced difference-all and the semi and anti-semi-joins, ranging from 1.2 seconds at the large memory to more than 9 seconds at the small memory (again, for Grace).

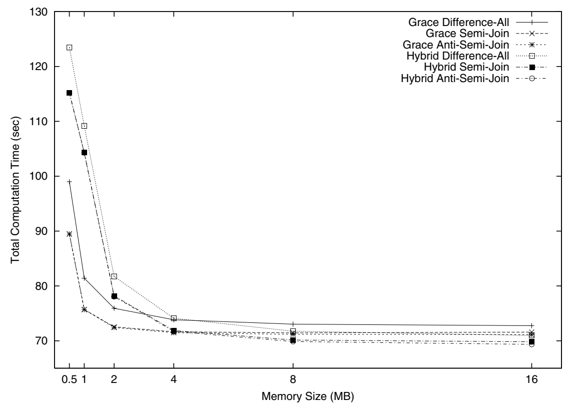

Figure 11. Total time over varying main memory.

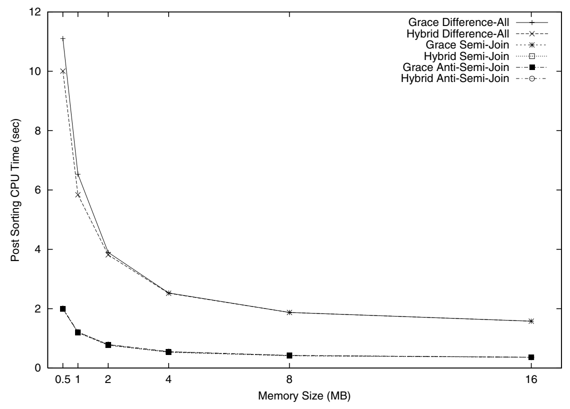

Figure 12. CPU costs over varying main memory.

In the data used for difference-all, there are eight duplicate values for each explicit attribute in the LHS (in the data for semi-join and anti-semi-join, there are no such duplicates in the LHS). All six algorithms effectively do a nested-loop join over the duplicate tuples, $1 \times 16 = 16$ LHS-RHS tuple pairs for semi-join and anti-semi-join and $8 \times 128 = 1024$ pairs for difference-all. This implies that for each LHS tuple, for these data sets, difference-all must examine 8 more RHS tuples. This completely explains the separate curve for difference-all.

We conclude from these experiments that the dominant cost of all of these algorithms is the sorting of both argument relations and outputing the resulting tuples. The algorithms consist primarily of a single scan of the two input relations, with minor variations on the seek time and CPU time required. The sequenced extensions do not incur additional intermediate I/O, nor significant additional processing time.

Algorithm	0.5MB			16MB		
	Seeks (#)	Seek Time (secs)	Other Time (secs)	Seeks (#)	Seek Time (secs)	Other Time (secs)
Grace Semi-Join	1853	18.5	71.0	9	0.1	71.5
Hybrid Semi-Join	4768	47.7	67.5	42	0.4	69.4
Grace Anti-Semi-Join	1861	18.6	72.0	9	0.1	71.0
Hybrid Anti-Semi-Join	4795	48.0	68.5	42	0.4	70.0
Grace Difference-All	1865	18.7	81.3	9	0.1	72.7
Hybrid Difference-All	4766	47.7	75.8	42	0.4	70.6

Table 2. Contribution of disk seeks during merge phase to overall time.

7. Summary and Future Work

In this paper we have discussed six temporal database operations: sequenced semi-join, sequenced anti-semi-join, sequenced difference, sequenced difference-all, sequenced intersection and sequenced intersection-all. The challenge is in dealing with the fragmentation of the left-hand periods of validity. In the worst case, a single tuple from the left-hand side can be fragmented into a number of pieces exceeding the cardinality of the right-hand size relation; we wish to permit this computation to run in restricted main memory buffers.

We showed how to express in SQL three of these operators, sequenced semi-join, anti-semi-join and difference-all; the other three can be implemented via the first three. For sequenced anti-semi-join and difference-all, the SQL statements are quite complex, requiring three or four joins and subqueries [Snodgrass 00]. Hence, we considered how these operators could be implemented inside a temporal DBMS.

We provided new algorithms for the three base operators, exploiting the sort order to emit result tuples as they are produced and to retain the needed state in the existing buffers. We implemented these three operators and measured their performance under a variety of conditions. In all cases, the execution time was that of a single conventional join, dominated by the cost of the sort, the initial read scane and the the cost of outputing the result tuples.

Our implementations for these operations are based on sort-merge join. However, our approach is not entirely dependent on the partitioning method used to collect tuples together [Graefe 93]; hash joins could be modified to implement these three algorithms. It would also be useful to consider parallel implementations of these operators.

It may be possible to adapt the difference-all approach to support intersection-all. The advantage of doing so is that two scans of the LHS would not be necessary, nor would writing out an intermediate relation, as is the case when intersection-all is implemented via two difference-all operations.

Finally, it would be interesting to implement *bitemporal* versions of these operators, sequenced in both valid and transaction time. [Böhlen et al. 00, pp. 449-451] gives an example in which bitemporal difference between a single LHS tuple and two RHS tuples generates eleven fragments in the result. It is not clear whether our approach of separate sort orders would apply; there is some theoretical evidence [Chomicki & Toman 98] that it would not.

8. Acknowledgements

We thank Dengfeng Gao and Stephen Kobourov for their help. This work was supported in part by NSF grants EIA-0080123 and IIS-0100436 and by a grant from the Boeing Corporation.

References

[Allen 83] J. F. Allen, "Maintaining Knowledge about Temporal Intervals," *Communications of the ACM* 26(11), pp 832–843, 1983.

[Bettini et al. 98] C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass and X. S. Wang, "A Glossary of Time Granularity Concepts," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia and S. Sripada (eds.), Springer, pp. 406–413, 1998.

[Böhlen et al. 96] M. H. Böhlen, R. T. Snodgrass and M. D. Soo, "Coalescing in Temporal Databases," in the *Proceedings of the International Conference on Very Large Databases*, pp. 180–191, Mumbai (Bombay) India, September, 1996.

[Böhlen et al. 00] M. H. Böhlen, C. S. Jensen and R. T. Snodgrass, "Temporal Statement Modifiers," *ACM Transactions on Database Systems*, 25(4):407–456, December 2000.

[Chomicki & Toman 98] J. Chomicki and D. Toman, "Temporal Logic in Information Systems," chapter 3 of **Logics for Databases and Information Systems**, J. Chomicki and G. Saake, editors, Kluwer, pp. 31–70, 1998.

[Edara & Gadia 93] M. Edara and S. Gadia. "Updates and Incremental Recomputation of Active Relational Expression in Temporal Databases," *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, pp. O1–O27, 1993.

[Etzion et al. 93] O. Etzion, A. Gal and A. Segev. "Temporal Active Databases," *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, pp. EE1–EE13, 1993.

[Graefe 93] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys* 25(2), pp. 73–170, 1993.

[Graefe et al. 94] Goetz Graefe, Ann Linville and Leonard D. Shapiro, "Sort vs. Hash Revisited," *IEEE Transactions on Knowledge and Data Engineering* 6(6):934–944, December, 1994

[Gunadhi & Segev 90] H. Gunadhi and A. Segev. "A Framework for Query Optimization in Temporal Databases," *Proceedings of the 5th International Conference on Statistical and Scientific Database Management*, pp. 131–147, 1990.

[Jensen et al. 92] C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev and R. T. Snodgrass. "A Glossary of Temporal Database Concepts," *ACM SIGMOD Record* 21(3), pp. 35–43, 1992.

[Kline & Soo 98] N. Kline and M. Soo, "The TIMEIT Temporal Database Testbed," 1998. `www.cs.auc.dk/TimeCenter/software.htm`.

[Leung & Muntz 90] T. Y. Leung and R. Muntz, "Query Processing for Temporal Databases," in *Proceedings of the 6th International Conference on Data Engineering*, Los Angeles, California, February 1990.

[Leung & Muntz 92] T. Y. Leung and R. Muntz, "Temporal Query Processing and Optimization in Multiprocessor Database Machines," in *Proceedings of the Conference on Very Large Databases*, August 1992.

[Li et al. 01] W. Li, D. Gao and R. T. Snodgrass, "Skew Handling Techniques in Sort-Merge Join," TIMECENTER Technical Report TR-62, June, 2001.

[Melton & Simon 93] J. Melton and A. R. Simon. **Understanding the New SQL: A Complete Guide**, Morgan-Kaufmann Publishers, San Francisco, CA, 1993.

[Mishra & Eich 92] P. Mishra and M. H. Eich. "Join Processing in Relational Databases," *ACM Computing Surveys*, 24:63–113, 1992.

[Segev & Gunadhi 89] A. Segev and H. Gunadhi, "Event-Join Optimization in Temporal Relational Databases," in *Proceedings of the Conference on Very Large Databases*, pages 205–215, August 1989.

[Snodgrass 87] R. T. Snodgrass, "The Temporal Query Language TQuel," *ACM Transactions on Database Systems*, (12)2:247–298, June, 1987.

[Snodgrass 00] R. T. Snodgrass, **Developing Time-Oriented Database Applications in SQL**, Morgan Kaufmann Publishers, San Francisco, 2000.

[Snodgrass et al. 96] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen and A. Steiner. International Organization for Standardization Proposal ANSI X3H2-96-501r1: "Adding Valid Time to SQL/Temporal," 1996.

[Soo et al. 94] M. D. Soo, R. T. Snodgrass and C. S. Jensen. "Efficient Evaluation of the Valid-Time Natural Join," *Proceedings of the Tenth IEEE International Conference on Data Engineering*, pp. 282–292, 1994.

[Tansel et al. 93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. T. Snodgrass. **Temporal Databases: Theory, Design, and Implementation**, Benjamin/Cummings Publishing Company, 1993.

[Zurek 96] T. Zurek, **Parallel Temporal Nested-Loop Joins**. Ph.D. Dissertation, Dept. of Computer Science, Edinburgh University, 1996.