Coalescing in Temporal Databases

Michael H. Böhlen and Richard T. Snodgrass and Michael D. Soo

April 27, 1997

TR-9

A TIMECENTER Technical Report

Title Coalescing in Temporal Databases

Copyright © 1997 Michael H. Böhlen and Richard T. Snodgrass and Michael D. Soo. All rights reserved.

Author(s) Michael H. Böhlen and Richard T. Snodgrass and Michael D. Soo

Publication History June 1996. Technical Report R-96-2026 Aalborg University
September 1996. Proceedings of the 22nd International Conference on Very Latge Data Bases.

Abstract

Coalescing is a unary operator applicable to temporal databases; it is similar to duplicate elimination in conventional databases. Tuples in a temporal relation that agree on the explicit attribute values and that have adjacent or overlapping time periods are candidates for coalescing. Uncoalesced relations can arise in many ways, e.g., via a projection or union operator, or by not enforcing coalescing on update or insertion. In this paper we show how semantically superfluous coalescing can be eliminated. We then turn to efficiently performing coalescing. We provide a variety of iterative and non-iterative approaches, via SQL and embedded SQL, that require no changes to the DBMS, demonstrating that coalescing can be formulated in SQL-89. Detailed performance studies show that all such approaches are quite expensive. We propose a spectrum of coalescing algorithms within a DBMS, based on nested-loop, explicit partitioning, explicit sorting, temporal sorting, and combined explicit/temporal sorting, as well as their hybrid variants. These algorithms are empirically compared, paying particular attention to attribute skew and timestamp distributions. The experiments show that coalescing can be implemented with reasonable efficiency, and with modest development cost.

1 Introduction

Coalescing [JCE$^+$94] is a unary operator applicable to temporal databases; it is similar to duplicate elimination in conventional databases. Temporal databases are extensions of conventional databases that support the recording and retrieval of time-varying information [TCG$^+$93]. Associated with each tuple in a temporal relation is a timestamp, denoting some period of time. In a temporal database, information is “uncoalesced” when tuples have identical attribute values and their timestamps are either adjacent in time (“meet” in Allen’s taxonomy [All83]) or share some time in common. Consider the relation in Figure 1. The tuples in this relation denote the fact “Ronald Reagan was president” over two adjacent time periods. The two tuples can be replaced by a single tuple, timestamped with the period [1981/01/20–1989/01/20), to represent when Ron was President, instead of which terms he was elected to, which is represented in the uncoalesced relation. In general, two tuples in a valid-time relation are candidates for coalescing if they have identical explicit attribute values, i.e., are *value-equivalent* [Sno87], and have adjacent or overlapping timestamps. Such tuples can arise in many ways. For example, a projection of a coalesced temporal relation may produce an uncoalesced result, much as duplicate tuples may be produced by a projection on a duplicate-free snapshot relation. In addition, update and insertion operations may not enforce coalescing, possibly due to efficiency concerns.

Name	Title	V
Ron	President	[1981/01/20–1985/01/20)
Ron	President	[1985/01/20–1989/01/20)

Figure 1: Uncoalesced Valid-Time Relation

As with duplicate elimination in snapshot databases, prior coalescing is necessary to ensure the semantics of some operators in temporal databases [SJS95], e.g., temporal aggregation [KSL95] and temporal selection [SSJ94]. While many temporal data models and languages have implicitly or explicitly assumed or provided coalescing [Ari86, BZ82, CC87, GV85, McK88, NA89, SSD87, Sar90a, Sno87, Tan86], only recently has the importance of this operation with respect to semantics and performance been emphasized [Böh94]. Consider the relation in Figure 1. A relational algebra expression that selects those persons who were president for more than six years does not return Ron because the valid-time of either fact is less than six years. However, if the relation is coalesced prior to the evaluation Ron qualifies. Thus, whether a relation is coalesced or not makes a semantic difference. In general, it is not possible to switch between a coalesced and an uncoalesced representation without changing the semantics of programs. Moreover, as frequently used database operations (projection, union, insertion, and update) may lead to potentially uncoalesced relations and because many (but not all) real world queries require coalesced relations, a fast implementation is imperative.

Coalescing is potentially more expensive than duplicate elimination, which relies on an equality predicate over the attributes. Coalescing also requires detecting tuple overlap, which is an inequality predicate over the timestamp attribute. Most conventional DBMSs handle inequality predicates poorly; the typical strategy is to resort to exhaustive comparison when confronted with such predicates [LM90], yielding quadratic complexity (or worse) for this operation, as will be demonstrated later in this paper. For these reasons, effective optimization techniques for temporal queries involving coalescing must be devised. Efficient algorithms for evaluating the coalescing operator are also needed. Together these capabilities are critical to achieving acceptable performance in a temporal DBMS. We address these topics in this paper.

The remainder of the paper is organized as follows. We first examine how coalescing has arisen in previous temporal data models and query languages. Section 3 formally defines the coalescing operator. Section 4 discusses how to eliminate semantically superfluous coalescing. We turn our attention to operator evaluation outside a DBMS in Section 5. In Section 6 we introduce seven algorithms to evaluate coalescing within the DBMS and empirically test them under a variety of database and system conditions. Finally, conclusions and directions for future work are offered in Section 7.

2 Related Work

Early temporal relational models implicitly assumed that the relations were coalesced. Ariav's Temporally Oriented Data Model (TODM) [Ari86], Ben Zvi's Time Relational Model [BZ82], Clifford and Croker's Historical Relational Data Model (HRDM) [CC87], Navathe's Temporal Relational Model (TRM) [NA89], and the data models defined by Gadia [GV85, Gad88], Sadeghi [SSD87] and Tansel [Tan86] all have this property. The term *coalesced* was coined by Snodgrass in his description of the data model underlying TQuel, which also requires coalescing [Sno87][1]. Later data models, such as those associated with HSQL [Sar90a] and TSQL2 [Sno95], explicitly required coalesced relations. The query languages associated with these data models generally did not include explicit constructs for coalescing. HSQL is the exception; it includes a `COALESCE ON` clause within the select statement, and an `COALESCE` optional modifier immediately following `SELECT` [Sar90b]. Some query languages that don't require coalesced relations provide constructs to specify coalescing; ChronoBase [Sri91], which provides a `max_holds` predicate, ChronoSQL [Böh94], and ATSQL2, a variant of TSQL2 which allows both duplicates and non-coalesced relations [BJS95], are examples.

For many of these query languages, temporal algebras have been defined [MS91]. For those based on attribute timestamping, projection retains coalescing; generally the algebras for these models extend the union operator so that it also guarantees coalescing [Tan86, McK88, Gad88]. For those models based on tuple timestamping, some also include coalescing in the projection operator, e.g., the conceptual algebra for TSQL2 [SJS95]. Navathe and Ahmed defined the first coalescing algebraic operator; they called this COMPRESS [NA89]. Sarda defined an operator called Coalesce [Sar90b], Lorentzos' FOLD operator includes coalescing [LJ88], Leung's second variant of a temporal select join operator TSJ_2 can be used to effect coalescing, and TSQL2's representational algebra also included a coalesce operator [SJS95].

The expressive power of coalescing has been open to question. Leung and Muntz state that "the `time-union` operator is really a fixed-point computation and cannot be expressed in terms of traditional relational algebra" [LM93, p. 337]. This implies that coalescing is beyond relational completeness; this is a commonly held belief. Recently, Leung and Pirahesh provided a mapping of the coalesce operation into *recursive* SQL [LP95, p. 329]. However, Lorentzos and Johnson provided a translation of his FOLD operator (which also incorporates coalescing) into Quel [LJ88, p. 295], implying that coalescing does not add expressive power to the relational algebra. In Section 5, we settle the question by proving that coalescing is expressible in the relational algebra. We will also show that performing coalescing solely within SQL or the relational algebra is impractical, due to its extremely poor performance.

Despite the need for effective query optimization techniques and operator evaluation algorithms for coalescing, there has been scant coverage in the literature on either of these topics (indeed, a contribution of the present paper is highlighting this operator for further investigation by the temporal database community). This is in contrast to the conventional duplicate elimination operator, for which a large body of research exists [Gra93]. Concerning temporal coalescing, Navathe and Ahmed provided the first algorithm: sort the relation on a composite key of explicit attributes and time start, then scan the relation, extending the period of some tuples and deleting other tuples [NA89]. Lorentzos uses a similar algorithm to implement FOLD [Lor93, p. 89]. In Section 6, we evaluate this algorithm against a suite of other approaches.

[1]SQL-92 contains an unrelated `COALESCE` operator that is shorthand of `CASE` that replaces NULL values with other values [MS93].

3 Basic Definitions

While coalescing can be defined in almost all temporal data models, we choose a particular representation in order to concretely define its semantics. Definitions for other data models, while not given here, can be constructed similarly.

The data model we use is a first normal form model that timestamps tuples with open periods $[t_s, t_e)$, i.e., an instant t is contained in $[t_s, t_e)$ if and only if contained in $t_s \le t < t_e$. In this paper, we consider *valid-time* relations [SA86, JCE$^+$94], modeling changes in the mini-world represented in the database, though our results apply equally to transaction-time relations, and can be extended to bitemporal relations.

We define a relation schema $R = (A_1, \ldots, A_N \| \text{VT})$ as a set of *explicit attributes* $\{A_1, \ldots, A_N\}$ and an period timestamp VT= $[S - E)$. We use r to denote an instance of R, and x and y to denote tuples in r. As a shorthand, we use A to represent the set of attributes $\{A_1, \ldots, A_N\}$.

Prior to defining the coalescing operator we first define three auxiliary predicates. The first predicate determines if two argument tuples agree on the values of their explicit attributes.

$$value\_equivalent(x, y) = (x[A] = y[A]).$$

The remaining predicates accept period timestamps as arguments. The second predicate determines if the beginning of the first argument period meets the ending of the second argument period, and vice-versa.

$$adjacent([S_1 - E_1), [S_2 - E_2)) = (E_1 = S_2 \vee S_1 = E_2).$$

The third predicate determines if two argument periods share any instant in time (termed a *chronon*), i.e., if the periods overlap.

$$overlap([S_1 - E_1), [S_2 - E_2)) = (\exists c((S_1 \le c < E_1) \wedge (S_2 \le c < E_2))).$$

Informally, a relation is coalesced if all pairs of tuples from the relation, excluding pairs of the same tuple, are either not value-equivalent, or, if they are value-equivalent, then they must be non-adjacent and non-overlapping.

Definition 3.1 Let $R = (A_1, \ldots, A_N \| VT)$ be a valid-time relation schema. An instance r of R is *coalesced* if and only if

$$\begin{aligned}\forall x \in r, y \in r(\quad & x \neq y \\ & \vee(\neg value\_equivalent(x, y)) \\ & \vee(\neg overlap(x[VT], y[VT]) \wedge \neg adjacent(x[VT], y[VT])))\end{aligned}$$

□

4 Eliminating Superfluous Coalescing

Due to the nature of coalescing (merging value-equivalent tuples), coalescing is at least as costly as duplicate elimination (deleting identical tuples) in non-temporal databases. This means that it is best to simply avoid coalescing where possible. As previously mentioned, coalescing is required at some places in order to guarantee the correctness of query results [Böh94, NA93, SJS95, Sri91]. However there are many cases where coalescing can be omitted or where it can be postponed.

4.1 Basic Assumptions

Whether or not coalescing can be omitted or delayed depends on the definition of temporal operations and on the basic framework of the optimizer. This situation is identical to duplicate elimination in non-temporal databases. It is well known that projection does (or has the potential to) introduce duplicates when evaluated on a relation with no duplicates, whereas a join does not. Obviously this is only true for a particular definition of the algebraic operators. Besides this it is important to know whether duplicates are possible in base relations.

We assume that temporal operations are implemented as extensions of standard relational database operations[2]. In the following, $t_i \circ t_j$ denotes the concatenation of two tuples whereas the selection condition c stands for a boolean condition on a tuple and the projection function f for a function that maps between tuples. Roughly, c corresponds to the where clause and f to the select clause of a SQL statement. We use a "^{v}" superscript to denote a valid-time operation.

Definition 4.1 The semantics of a period-based temporal algebra is defined as follows.

$$\begin{array}{lcl}
\sigma_c^v r & \triangleq & \{\langle t \| [S-E) \rangle \mid \langle t \| [S-E) \rangle \in r \wedge c(\langle t \| [S-E) \rangle)\} \\
\pi_f^v r & \triangleq & \{\langle t_1 \| [S-E) \rangle \mid \langle t \| [S-E) \rangle \in r \wedge t_1 = f(\langle t_2 \| [S-E) \rangle)\} \\
r_1 \cup^v r_2 & \triangleq & \{\langle t \| [S-E) \rangle \mid \langle t \| [S-E) \rangle \in r_1 \vee \langle t \| [S-E) \rangle \in r_2\} \\
r_1 \times^v r_2 & \triangleq & \{\langle \langle t_1, [A-B) \rangle \circ \langle t_2, [C-D) \rangle \| [S-E) \rangle \mid \\
 & & \qquad \langle t_1 \| [A-B) \rangle \in r_1 \wedge \langle t_2 \| [C-D) \rangle \in r_2 \wedge \\
 & & \qquad S = \max(A, C) \wedge E = \min(B, D) \wedge S < E\} \\
r_1 \setminus^v r_2 & \triangleq & \{\langle t \| [S-E) \rangle \mid \langle t \| [A-B) \rangle \in r_1 \wedge \\
 & & \qquad \exists C(\langle t \| [C-S) \rangle \in r_2 \wedge A \leq S \vee S = A) \wedge \\
 & & \qquad \exists D(\langle t \| [E-D) \rangle \in r_2 \wedge B \geq E \vee E = B) \wedge S < E \wedge \\
 & & \qquad \neg \exists U, V(\langle t \| [U-V) \rangle \in r_2 \wedge V > S \wedge U < E)\}
\end{array}$$

□

Temporal selection is a straightforward generalization of non-temporal selection. The same is true for temporal projection and temporal union. A temporal Cartesian product computes the intersection of the respective valid-times whereas temporal set difference subtracts periods. As with the relational algebra, additional temporal operators, e.g., contains join, can be defined in terms of these basic operators.

While all operations are *period-based*, i.e., the format of periods is relevant for the computation of the result, it is also the case that these operators respect *snapshot reducibility* [BJS95, Sno87]. This means that one can define the semantics of a temporal operation in terms of its non-temporal counterpart applied to all snapshots of a database. But note that while, at the conceptual level, these operators simultaneously operate on many states of the databases, only start and end points of periods are considered, never intermediate time points. This allows for an efficient (granularity-independent) implementation.

Based on Definition 4.1 we identify those operations that potentially destroy coalescing and those that preserve coalescing.

Theorem 4.1 Temporal projection and temporal union have the potential to return an uncoalesced relation when evaluated over coalesced input relations.

Proof: Assume temporal schemas $R_1 = (A, B \| VT)$ and $R_2 = (A, B \| VT)$ with associated coalesced relation instances $r_1 = \{\langle a, b \| [2-5) \rangle, \langle a, c \| [5-8) \rangle\}$ and $r_2 = \{\langle a, b \| [5-9) \rangle\}$. According to Definition 4.1 we have $\pi_A^v(r_1) = \{\langle a \| [2-5) \rangle, \langle a \| [5-8) \rangle\}$ and $r_1 \cup r_2 = \{\langle a, b \| [2-5) \rangle, \langle a, c \| [5-8) \rangle, \langle a, b \| [5-9) \rangle\}$. Both results are uncoalesced because they contain value-equivalent tuples with adjacent periods (c.f., Definition 3.1). □

Theorem 4.2 Temporal selection, temporal Cartesian product, and temporal negation preserve coalescing when evaluated over coalesced input relations.

Proof: Temporal selection selects a subset of the input tuples. The input relation does not contain value-equivalent tuples with overlapping or adjacent periods which is also true for any subset of the relation. The proofs for temporal Cartesian product and temporal negation are similar. In both cases (see below) the valid-time of the tuples of the input relation r_1 is *narrowed*, i.e., restricted to a sub-period of the original period. As r_1 does not contain value-equivalent tuples with adjacent or overlapping periods this is also true for any relation that only contains tuples with a valid-time contained in the valid-time of a value equivalent tuple in r_1. It's straightforward

[2]While we couch our discussion in terms of a particular temporal algebra, a similar analysis could be performed on other temporal algebras [MS91].

to see that the output relation of temporal negation satisfies this criterion. With temporal Cartesian product we first abstract the result tuple to $\langle t_1, \ldots \| [S{-}E] \rangle$. Assuming that there are no additional explicit attributes the result relation qualifies again. By adding explicit attributes we don't get more value-equivalent tuples. At best, i.e., if for each tuple the additional explicit attribute values are the same, we get the same number of value-equivalent tuples. □

These theorems can be extended to apply to derived operators. For example, a contains join can be defined in terms of temporal Cartesian product (which retains the timestamps of the underlying tuples) and temporal selection, and thus preserves coalescing.

We discuss coalescing by partitioning the set of coalescing rules into two classes: unconditional and conditional ones. Unconditional coalescing rules only depend on temporal relational operators, whereas conditional coalescing rules additionally depend on parameters to these operators (e.g., selection conditions). Conditional rules are harder to deal with because they involve the analysis of functions and boolean expressions.

4.2 Unconditional Coalescing Rules

A first rule eliminates successive coalescing operations.

$$\text{(r0)} \quad coal(coal(r_1)) \equiv coal(r_1)$$

A more enhanced set of optimization rules exploits the fact that some operations preserve coalescing.

$$\begin{array}{lll} \text{(r1)} & coal(r_1 \times^v r_2) & \equiv coal(r_1) \times^v coal(r_2) \\ \text{(r2)} & coal(r_1 \setminus^v r_2) & \equiv coal(r_1) \setminus^v coal(r_2) \\ \text{(r3)} & coal(\sigma_c^v(coal(r_1))) & \equiv \sigma_c^v(coal(r_1)) \end{array}$$

Temporal Cartesian product and temporal set difference both preserve coalescing and they are invariant with respect to the timestamp format of the input relations. Note that it is not a priori clear whether to push coalescing inside or whether to defer it. For example, if r_1 and r_2 are both base relations known to be coalesced (e.g., because of model inherent constraints or according to database statistics) we push coalescing inside. In this case rule (r1) degenerates to $coal(r_1 \times^v r_2) \equiv r_1 \times^v r_2$ and we do not have to coalesce at all. However if r_1 and r_2 are uncoalesced base relations and if a join (i.e., a Cartesian product followed by a selection) is expected to cut down on the size of the input relations it might be better to postpone coalescing until after the join (see also (r6), below).

Analogous to these rules is rule (r4) which states that it is unnecessary to coalesce before and after temporal union.

$$\text{(r4)} \quad coal(coal(r_1) \cup^v coal(r_2)) \equiv coal(r_1 \cup^v r_2)$$

This is quite obvious because temporal union potentially destroys coalescing but is invariant with respect to the timestamp format of input relations. Note that it is not possible to give a similar rule for temporal projection π_f^v, the other operation that potentially destroys coalescing. The reason for this is that the result of a temporal projection may vary with the timestamp format of the input relation (if the projection function f computes a value based on the valid-time of the input relation).

A final unconditional optimization applies to temporal set difference.

$$\text{(r5)} \quad r_1 \setminus^v coal(r_2) \equiv r_1 \setminus^v r_2$$

Rule (r5) states that it is not necessary to coalesce the negated operand in a temporal set difference. This result is in accordance with non-temporal set difference which is indifferent with respect to duplicates in the negated part.

4.3 Conditional Coalescing Rules

As stated earlier unconditional coalescing rules are easier to deal with than conditional rules. However restricting attention to this class of rules means not to exploit the full potential of query optimization. Enhanced query optimizers have to take conditional coalescing rules into consideration as well. A first conditional coalescing rule states that coalescing can be deferred until after a selection if the selection condition c does not constrain the valid-time of the input relation, i.e., if $independent\_of\_vt(c)$ is true.

$$\text{(r6)} \quad \sigma_c^v(coal(r_1)) \equiv coal(\sigma_c^v(r_1)) \quad \text{iff } independent\_of\_vt(c)$$

Note that this rule can be quite effective in terms of efficiency, when the selectivity of the predicate is high. Given that the costs of coalescing are super-linear with the number of tuples to be coalesced (as shown in Sections 5 and 6) it can be useful to postpone coalescing.

A similar rule applies to the temporal projection operator.

$$\text{(r7)} \quad coal(\pi^v_f(coal(r_1))) \equiv coal(\pi^v_f(r_1)) \quad \text{iff } independent\_of\_vt(f)$$

Finally, there is a rule that eliminates coalescing of coalesced (base) relations.

$$\text{(r8)} \quad coal(r_1) \equiv r_1 \quad \text{iff } is\_coalesced(r_1)$$

In data models that guarantee that base relations are coalesced this rule is quite effective. Models without this restriction still can exploit the rule by maintaining appropriate statistics.

5 Performing the Coalescing Operation

In this section we investigate the possibilities available to a *database user* to implement coalescing. A database user cannot directly access and manipulate physically stored tables. Instead he is forced to use the data manipulation interface (e.g., SQL) to do changes. As we will see this is a significant obstacle. In Section 6 we consider implementing coalescing within a DBMS.

We assume a valid-time relation r with an explicit attribute c. Valid-time is represented by two attributes, `S`, denoting the start point and `E`, denoting the end point.

```
CREATE TABLE r(S INTEGER NOT NULL, E INTEGER NOT NULL, c INTEGER NOT NULL)
```

Figure 2 shows two instantiations of r, each with n tuples, which we have used to run our tests. (See Section 6 for a discussion of different database instances.) The first one, r_1, consists of value-equivalent tuples with adjacent valid-times. All tuples in r_1 can be coalesced into a single tuple with $S = 0$ and $E = n$, i.e., the reduction factor [Gra93, p.100] is n. Relation r_2, on the other hand, is already coalesced, thus, the reduction factor is 1.

r_1

S	E	c
0	1	0
1	2	0
2	3	0
...	...	...
$n-1$	n	0

r_2

S	E	c
0	1	0
1	2	1
2	3	2
...	...	...
$n-1$	n	$n-1$

Figure 2: Two instantiations of r which we use to run performance measurements.

5.1 SQL Implementation

All database statements we give are specific to Oracle. The tests have first been run on a Sun 3/80 running Oracle 6.2.1. Later we have rerun them on a Sparc 5 running Oracle 7.0.16. While the absolute numbers were quite different the relative differences remained about the same.

5.1.1 Iterative Approaches

Coalescing requires (chains of) value-equivalent tuples with adjacent or overlapping valid-times to be coalesced into a single tuple [Böh94]. A similar problem is the computation of the transitive closure of a graph with the subsequent deletion of non-maximal paths. In SQL the computation of the transitive closure can be implemented by iterating an insert statement that coalesces two valid-time periods (i.e., paths) and inserts a new tuple into the relation. An optimization exploits the fact that we are only interested in maximal periods. Therefore rather than inserting a new tuple (and retaining the old ones) we update one of the tuples that was used to derive the new one. This approach can be implemented by iterating an update statement. In each case, the statement (`INSERT` or `UPDATE`) is repeated until `r` doesn't change.

```
INSERT INTO r
  SELECT l.S, h.E, l.c
    FROM r l, r h
    WHERE l.c = h.c
    AND l.S < h.S
    AND h.S <= l.E
    AND l.E < h.E
  MINUS SELECT * FROM r
```

```
UPDATE r l
SET (l.E) = (SELECT MAX(h.E)
               FROM r h
               WHERE l.c = h.c
               AND l.S < h.S
               AND l.E >= h.S
               AND l.E < h.E)
WHERE EXISTS (SELECT *
              FROM r h
              WHERE l.c = h.c
              AND l.S < h.S
              AND l.E >= h.S
              AND l.E < h.E)
```

The basic idea is the same for both statements. We search for value-equivalent tuples with adjacent or overlapping valid-times. The insert statement then coalesces these tuples and computes a new one that is inserted into the table if it is not there already. The update statement updates one of the existing tuples rather than inserting a new one. The complexity of the statements indicates that it is less expensive to determine tuples to be inserted than it is to determine tuples to be updated. However as we will see in Section 5.1.4 this advantage of the insert statement is more than compensated for by the costs to handle new data (i.e., the data to be inserted).

After both kinds of iterations we have to delete tuples with non-maximal valid-times (i.e., with valid-times that are contained in the valid-time of another value-equivalent tuple).

```
DELETE FROM r a0
  WHERE EXISTS (SELECT *
                FROM r a1
                WHERE a0.c = a1.c
                AND (a0.S > a1.S AND a0.E <= a1.E OR
                     a0.S >= a1.S AND a0.E < a1.E))
```

If a database system supports recursion or transitive closure computations it is possible to perform the iteration directly in SQL (c.f., [LP95]) instead of embedding SQL into a general purpose programming language.

5.1.2 Non-Iterative Approaches

The algorithms developed in the previous section where based on ideas used for the computation of transitive path closures. However time has special properties which makes it possible to employ quite different algorithms. Assuming that time is linear, i.e., totally ordered, it is possible to compute maximal periods with a single SQL statement (see also [Cel95, p. 291]). The basic idea is illustrated by the following range-restricted first-order logic formula.

$$\begin{array}{l} r(\langle X \| [S-\_) \rangle) \land r(\langle X \| [\_-E) \rangle) \land S < E \land \\ \forall A(r(\langle X \| [A-\_) \rangle) \land S < A < E \rightarrow \exists U, V(r(\langle X \| [U-V) \rangle) \land U < A \leq V)) \land \\ \neg \exists A, B(r(\langle X \| [A-B) \rangle) \land (A < S \leq B \lor A \leq E < B)) \end{array}$$

On the first line we search for two (possibly the same) value-equivalent tuples defining start point S and end point E of a coalesced tuple. The second line ensures that all start points A between S and E of value-equivalent tuples are extended (towards S) by another value-equivalent tuple. This guarantees that there are no holes between S and E, i.e., no time points where the respective fact does not hold. The last line makes sure that we get maximal periods only, i.e., S and E may not be part of a larger value-equivalent tuple.

The above first-order logic formula can be translated to SQL by first eliminating ∀-quantifiers and implications and then translating to SQL directly (c.f., [BCST96]).

```
SELECT DISTINCT f.S, l.E, f.c
FROM r f, r l
WHERE f.S < l.E
AND f.c = l.c
AND NOT EXISTS (SELECT *
                FROM r m
                WHERE m.c = f.c
                AND f.S < m.S AND m.S < l.E
                AND NOT EXISTS (SELECT *
                                FROM r a1
                                WHERE a1.c = f.c
                                AND a1.S < m.S AND m.S <= a1.E))
AND NOT EXISTS (SELECT *
                FROM r a2
                WHERE a2.c = f.c
                AND (a2.S < f.S AND f.S <= a2.E OR
                     a2.S <= l.E AND l.E < a2.E))
```

The select statement is considerably more complex than the insert or update statement. However it only has to be executed once and does not require a procedural extension of SQL or use of recursive constructs. In Section 5.1.4 we will see how these aspects impact performance.

5.1.3 Optimizations

There exist standard optimization techniques that might be applied to either the iterative or the non-iterative solutions [O'N94]. It is possible to create an index for the purpose of coalescing.

```
CREATE INDEX i1 ON r(S)
```

The effectiveness of such an optimization depends on the query optimizer of the respective database system. Therefore we only include them selectively into the empirical measurements, mainly to exemplify the relative speedup that can be expected. It should be obvious that standard optimization techniques, like indexing and clustering, are applicable to all three approaches provided above. However their effectiveness has to be verified in each particular situation. We show the results for the most promising algorithm only.

5.1.4 Empirical Results

Figure 3 summarizes the performance of the three approaches, along with the update algorithm using an index. All were performed on the uncoalesced relation (r_1 in Figure 2). The x-axis provides the size of the relation; the y-axis the total time for coalescing, in seconds. Recall that for all but the "select" approach the statement has to be iterated (in this, worst, case, $\log_2(n)$ times [Böh94]) until the fixpoint is reached, i.e., until no more tuples are inserted or updated (Oracle provides an easy way to ascertain this). Timings with an index are included for the update statement only, which has the best overall performance. Here, the index has little impact on the performance.

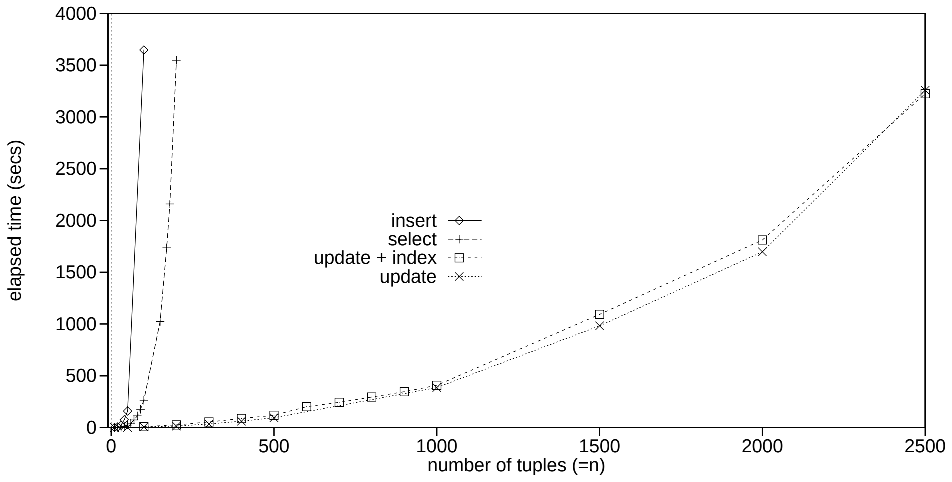

Figure 3: Coalescing a relation where all tuples can be coalesced into a single one.

The costs to coalesce relation r_1 are dominated a) by the costs to insert new tuples and b) by the complexity of the SQL statement. It also turns out to be much cheaper to iterate the moderately complex update statement instead of executing the even more complex select statement once. Even an index does not help. Instead of speeding up the select statement an index slows it down even more. Also the update statement could not take advantage of an index. Experiments with indexes revealed that it is best to create a temporary index on the valid-time start point for the purpose of the iteration. A permanent index considerably slows down the delete statement. The increase in execution time was always greater than the time to create and drop the index. Finally, we note that performance would perhaps be improved significantly if Oracle supported *local tables*, as proposed for SQL3. Such tables, which persist only for the duration of a single transaction, would incur significantly smaller penalties of update and insertion, as these modifications would not have to be logged or participate in locking.

Figure 4 displays the costs to coalesce an already coalesced relation. Note that the scale of the x-axis has

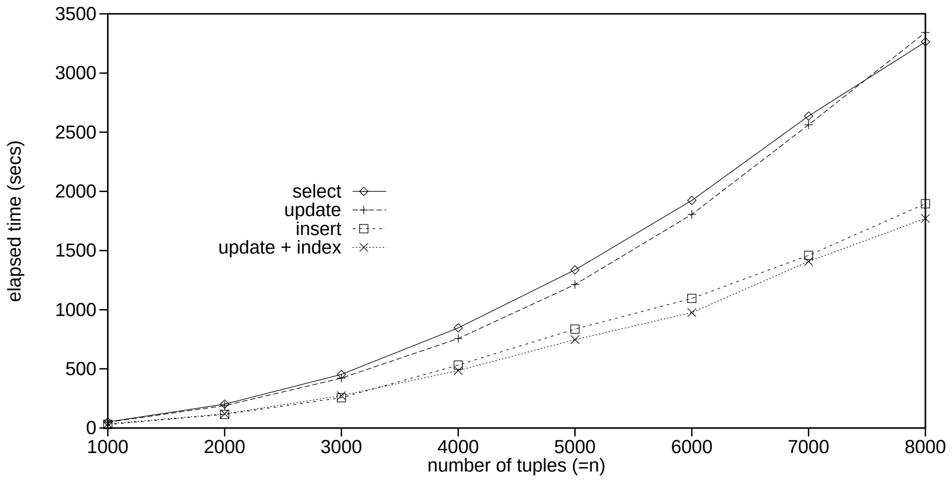

Figure 4: Coalescing a relation without value-equivalent tuples.

changed. When the relation is already coalesced, the coalescing algorithms all are still very slow, with the top two about 70% more costly than the bottom two. Not surprisingly the insert statement is the most efficient. Even without an index it achieves a performance comparable to that one of the update statement with an index. Clearly an index would speed up the insert statement even more. However we have not included these measurements because of the poor performance of the insert statement when the relation contains value-equivalent tuples.

In summary, performing coalescing in SQL, using any of the approaches discussed, is exceedingly inefficient.

5.2 Main Memory Implementation

One means to improve the performance of coalescing is to load the relation into main memory, coalesce it manually, and then store it back in the database. A straightforward implementation suffers from two serious constraints. It is not always feasible to load a (huge) relation into main memory and coalescing is based on sorting which is time-consuming and non-trivial. Of course, both issues can be addressed with a sophisticated implementation. However, this means that we have to re-implement DBMS functionality. A better approach is to fetch tuples ordered primarily by explicit attribute values and secondarily by start time. This allows us to reuse the sorting mechanism of the DBMS and to perform coalescing with just a single tuple in main memory. The core of the C code of the coalescing algorithm is displayed below. (The calls `oparse`, `oexec`, and `ofetch` are OCI (Oracle Call Interface) procedures.)

```
/* open a cursor to fetch tuples from the DB */
oparse(cda1, "SELECT S, E, c FROM r ORDER BY c, S");
oexec(cda1);

/* open a cursor to store tuples back in the DB */
oparse(cda2, "INSERT INTO r_coal VALUES (:1, :2, :3)");

/* main memory coalescing */
```

```
while (ofetch(cda1) == 0) { /* fetch all tuples */
  if (curr_tpl.c == next_tpl.c && next_tpl.S <= curr_tpl.E) {
    /* value-equivalent and overlapping */
    if (next_tpl.E > curr_tpl.E) curr_tpl.E = next_tpl.E;
  } else {
    /* not value-equivalent or non-overlapping */
    oexec(cda2); /* store back current tuple */
    curr_tpl.S = next_tpl.S;
    curr_tpl.E = next_tpl.E;
    curr_tpl.c = next_tpl.c;
  }
}
oexec(cda2); /* store back current tuple */
```

The main memory approach has one disadvantage when compared to the SQL implementations in the previous section. It suffers from the so-called "entry-costs", i.e., the costs to move data between the database and the application. Our measurements reveal that the entry-costs are indeed significant. However, they are considerably lower than the costs to execute the SQL statements discussed in the previous section.

Figure 5 illustrates the costs to coalesce the template relations in Figure 2. It is cheaper to apply main memory coalescing to relations with large reduction factors. (This is in contrast to the SQL-based algorithms discussed in

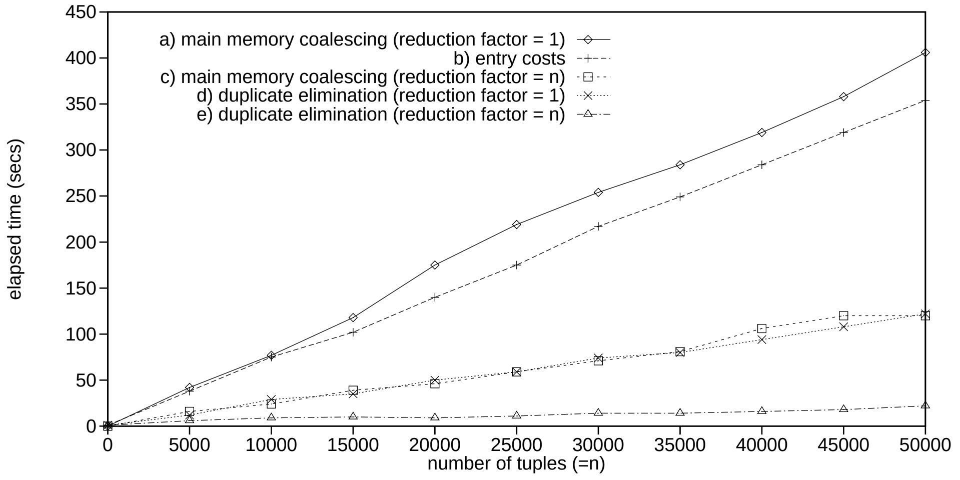

Figure 5: Coalescing a relation by loading it into main memory and then storing it back, as compared with the entry costs and duplicate elimination.

the previous section.) The reason is that less tuples have to be stored back. Four factors contribute to the total coalescing time, namely the sorting performed by the DBMS, loading the relation into main memory, the main memory coalescing steps, and storing back the tuples into the database. The dominating factor are the entry costs (storing and fetching tuples). The additional costs for coalescing (DBMS sorting and main memory operations) are the difference between a) and b), the top two graphs. They are not relevant. The costs for storing back are the dominating factor. These costs are the difference between a) and c). They amount to about 70% of the time. Note that a) is the upper bound for the coalescing costs (maximal costs for storing back, reduction factor = 1) whereas c) is the lower bound (no costs for storing back, reduction factor = n). Finally, we recall that, as with the SQL-based algorithms, performance might improve significantly if local tables were available. Specifically, we expect them to speed up the storing back into the database.

Also shown in the figure is the cost to eliminate duplicates. Note that in both cases (with and without duplicates), duplicate elimination within the DBMS is much faster than coalescing outside the DBMS. This is an apples and oranges comparison, as coalescing is somewhat more complex than duplicate elimination. Nevertheless, it indicates the potential performance improvement possible by implementing coalescing within the DBMS, which we examine in detail in the next section.

In summary, a database user should do coalescing by fetching all tuples ordered primarily by explicit attribute values and secondarily by the start time. A single tuple is kept in main memory. Whenever this tuple cannot be coalesced with a newly fetched tuple anymore it is stored back.

6 DBMS Implementation

In this section, we derive new algorithms to implement coalescing as an internal DBMS operation, and describe a performance study comparing these algorithms under a variety of database conditions. Our goals are two-fold: to identify the space of algorithms applicable to coalescing, and to investigate how coalescing can be implemented cheaply, and with adequate performance.

6.1 Algorithms

Operationally, coalescing is very similar to unary relational operations such as duplicate elimination, and related operations such as grouping for aggregation. Whereas duplicate elimination matches value-equivalent tuples, coalescing performs the same matching, with the added restriction that tuple timestamps must either be overlapping or adjacent. We use this similarity to derive coalescing algorithms from well-established techniques for duplicate elimination.

Duplicate elimination algorithms, and all relational query evaluation algorithms, are based on three main paradigms: nested-loop, partitioning, and sort-merge [Gra93]. Nested-loop algorithms are the the simplest; typical implementations perform exhaustive comparison to find matching input tuples. Partitioning and sorting are divide and conquer algorithms that preprocess the input in order to reduce the number of comparisons needed to find matching tuples.

Partition-based duplicate elimination divides the input tuples into buckets using the attributes of the input relation as key values. Each bucket contains all tuples that could possibly match with one another, and the buckets are approximately the size of the alloted main memory. The result is produced by performing an in-memory duplicate elimination on each of the derived buckets.

Sort-merge duplicate elimination 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 duplicates encountered during the merge step are eliminated.

We adapt these basic duplicate elimination algorithms to support coalescing. To enumerate the space of coalescing algorithms, we use the duality of partitioning and sort-merge [GLS94]. 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 duplicate elimination algorithms order the input relation on the relation's explicit attributes. For a temporal relation, which has timestamp attributes in addition to explicit attributes, there are four possibilities for ordering the relation. First, the relation can be sorted using the explicit attributes exclusively. Second, the relation can be ordered on time, using either the starting or ending timestamp [LM93, Seg93]. The choice of starting or ending timestamp dictates an ascending or descending sort order, respectively. Third, the relation can be ordered primarily on the explicit attributes and secondarily on time [NA93]. Lastly, the relation can be ordered primarily on time and secondarily on the explicit attributes.

By duality, the division step of partition-based algorithms can partition using any of these options [LM93, Seg93]. Hence, four choices exist for the dual steps of merging in sort-merge or partitioning in partition-based methods.

Lastly, it has been recognized that the choice of buffer allocation strategy, Grace or hybrid [DKO^{+}84], is independent of whether a sort or partition-based approach is used [Gra93]. Hybrid policies minimize the flushing of intermediate buffers from main memory, and hence can decrease the I/O cost for a given execution.

Figure 6 shows the choices of sort-merge versus partitioning, the possible sorting/partitioning attributes, and the possible buffer allocation strategies. Combining all possibilities gives sixteen possible evaluation algorithms. Including the basic nested-loop algorithm results in a total of seventeen possible algorithms. The seventeen algorithms are named and described in Figure 7.

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

Figure 6: Space of possible evaluation algorithms

Algorithm	Name	Description
Explicit sort	ES	Grace sort-merge on explicit attributes
Hybrid explicit sort	ES-H	Hybrid sort-merge on explicit attributes
Temporal sort	TS	Grace sort-merge on timestamps
Hybrid temporal sort	TS-H	Hybrid sort-merge on timestamps
Explicit/temporal sort	ETS	Grace sort-merge on explicit attributes/time
Hybrid explicit/temporal sort	ETS-H	Hybrid sort-merge on explicit attributes/time
Temporal/explicit sort	TES	Grace sort-merge on time/explicit attributes
Hybrid temporal/explicit sort	TES-H	Hybrid sort-merge on time/explicit attributes
Explicit partitioning	EP	Grace partition on explicit attributes
Hybrid explicit partitioning	EP-H	Hybrid partition on explicit attributes
Temporal partitioning	TP	Range partition on time
Hybrid temporal partitioning	TP-H	Hybrid range partition on time
Explicit/temporal partitioning	ETP	Grace partition on explicit attributes/time
Hybrid explicit/temporal partitioning	ETP-H	Hybrid partition on explicit attributes/time
Temporal/explicit partitioning	TEP	Grace partitioning on time/explicit attributes
Hybrid temporal/explicit partitioning	TEP-H	Hybrid partitioning on time/explicit attributes
Nested-loop	NL	Exhaustive matching

Figure 7: Possible algorithms for performance study

Of the seventeen possible choices, we excluded six of the algorithms, TES, TES-H, TP-H, ETP, ETP-H, TEP, and TEP-H from the final study. TES and TES-H are optimizations of TS and TS-H, respectively, using a secondary sort-order on the explicit attributes. Intuitively, a secondary ordering on explicit attributes would not be effective if the start time of value-equivalent tuples are separated by long time periods. TP-H, ETP, ETP-H, TEP, and TEP-H perform partitioning on time, either primarily (TP, TP-H, TEP and TEP-H) or secondarily (ETP and ETP-H). For each of these algorithms, range-partitioning [LM91] is performed on the period timestamp attributes. Whereas range partitioning has been successfully applied in conventional query evaluation with its discrete attribute values [DNS91], it is much more difficult to perform range-partitioning using more complex period timestamps. We included the simplest temporal partitioning algorithm, TP, in the study.

The algorithms we include are adaptations of existing duplicate elimination algorithms. A database vendor can use these techniques to construct coalescing operators from existing code at fairly minimal implementation cost. While we do not consider them in this study, optimizations such as read-ahead using forecasting, early coalescing, merge optimizations, large cluster sizes, and bucket tuning [Gra93] can be applied to the coalescing algorithms as well.

A final few words of explanation are needed. We used a simple implementation of hybrid buffer management. For partition-based algorithms, e.g., EP-H, a partition was chosen to remain memory resident without being flushed to disk during, and after, the division step. Similarly, for the sort-based algorithms, ES-H, TS-H, and ETS-H, most of the last run generated was retained in memory rather than being flushed to disk. This required fairly straightforward calculations to allocate some of the buffer space being used by the last run to the remaining runs during merging.

For the algorithms ES, ES-H, TS, TS-H, EP, TP, EP-H, and NL we built the temporal element [Gad88] of value-equivalent tuples as a main-memory data structure. The temporal element was represented as a modified binary tree, where nodes in the tree contained a time period and a left and a right child pointer. Partial coalescing

Parameter	*Value*
Relation size	16 MB
Tuple size	16 bytes
Tuples per relation	1 M
Timestamp size ([s,e])	8 bytes
Explicit attribute size	8 bytes
Relation lifespan	100000 chronons
Page size	1 KB
Cluster size	32 KB

Figure 8: System characteristics

Parameter	*Value*
Sequential I/O cost	5 msec
Random I/O cost	25 msec
Explicit attribute compare	2 μsec
Timestamp compare	4 μsec
Pointer compare	1 μsec
Pointer swap	3 μsec
Tuple move	4 μsec

Figure 9: Cost metrics

was performed on insertion, and maximal periods, as required by coalescing, were produced when the tree was traversed, i.e., after all value-equivalent tuples had been scanned. In all cases, the space requirements of the temporal element was small relative to the available buffer space. The algorithms ETS and ETS-H did not require this data structure since the secondary sort order on time allows a constant in-memory workspace [LM93].

The temporal sorting algorithms, TS and TS-H, use tuple caching [SSJ94] to retain, in memory, tuples during the merging step that could coalesce with tuples appearing later in the scan. The tuple cache size was set at 32 K, i.e., one cluster of I/O (see Figure 8). The temporal partitioning algorithm, TP, which could also use tuple caching, was instead implemented using simple tuple replication.

Lastly, all algorithms were developed and experiments were run using the TIME-IT temporal database test environment [KS95], a system for prototyping query evaluation components. TIME-IT provides a synthetic temporal database generator, a simulated single disk system, and I/O and CPU cost measurement tools.

6.2 Parameters

Using TIME-IT we fixed several parameters describing all test relations used in the experiments. These parameters and their values are shown in Figure 8. (The page size of 1 K is fixed by TIME-IT. We plan to enhance the tool to support variable page sizes, but, in any case, our present results would scale to large page sizes.) We fixed the tuple size at 16 bytes and the relation size at 16 MB, giving 1 M tuples per relation. We chose a 16 M relation size since we were less interested in absolute size than in the ratio of input size to available main memory. 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 we used for all experiments are shown in Figure 9. We measured both main memory operations and disk I/O operations. All operations were measured synthetically using facilities provided by TIME-IT to eliminate any undesired system effects from the results. For disk operations, random and sequential access were measured separately with a five times cost factor for random accesses. 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.

6.3 Simple Experiments

In this section, we mimic the experiments of Section 5, by testing the algorithms when no coalescing occurs, and when all tuples coalesce into a single tuple. These experiments correspond to duplicate elimination where the reduction factor is one and the input cardinality, respectively.

6.3.1 No Coalescing

Analogous to the experiments in Section 5, we devised a "worst-case" experiment where essentially no coalescing occurs. We generated a single relation with randomly distributed explicit attribute values and randomly distributed period timestamps. The explicit attributes were two integer attributes drawn from the range 0 to $2^{31} - 1$, giving $(2^{31})^2$ different possibilities for the 1 M tuples. Period timestamps were set to be one chronon in duration, though, timestamping with longer periods would still result in little coalescing due to the lack of value-equivalence. Each algorithm was run on the relation using six memory allocations, from 0.5 M to 16 M, for an effective memory to input ratio of 1:32 at the smallest memory allocation and 1:1 at the largest. The results are shown in Figure 10. Notice that both the x-axis and y-axis are log-scaled.

Graphs in this section are organized as follows. The same line/point style is used for a given algorithm consistently throughout all graphs in this section. Algorithms appear in a particular graph legend according to their top-to-bottom order at the left margin of the graph. Lastly, corresponding pairs of Grace/hybrid algorithms are plotted in the same linestyle, but with different points.

Nested-loop is clearly not competitive in Figure 10. We implemented a standard block-oriented nested-loop algorithm. However, as noted in Section 5, is not sufficient to make a simple quadratic scan of the input. Like the SQL solutions of Section 5, the nested-loop program is effectively performing a fixpoint computation.

A simple improvement to this algorithm is to first sort the input on the explicit attributes, and use the sort-ordering to reduce the number of blocks scanned in the inner loop. This is the essentially the approach proposed by Navathe and Ahmed for their COMPRESS operator [NA93] and by Lorentzos for his Fold operator [Lor93]. However, rather than perform a separate sort operation, it is possible to sort and coalesce simultaneously. The sort-merge algorithms we consider operate in this manner. As can be seen from the graph, ES has a nearly 1000% advantage over NL at the smallest memory size, and is uniformly superior across all memory sizes. We will not consider nested-loop in the remainder of this paper.

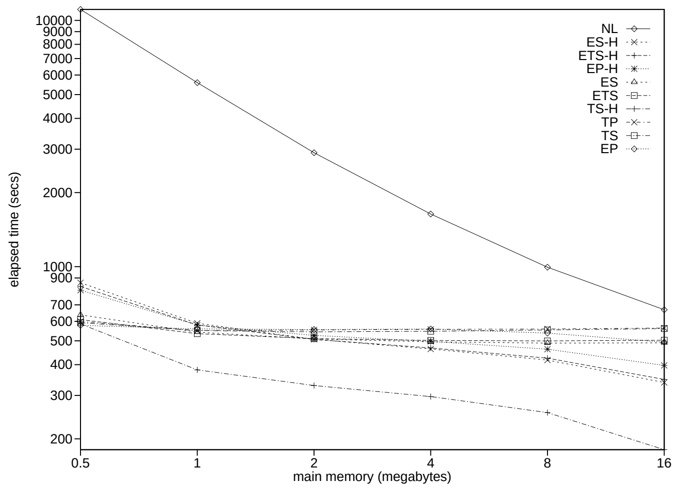

Figure 10: No coalescing (reduction factor = 1)

In order to get a better picture of the performance of the remaining algorithms, we plot them separately in Figure 11. Again both the x-axis and y-axis are log-scaled. As can be seen, the results closely agree with those from duplicate elimination: corresponding sort-based and partition-based algorithms have largely identical performance; careful tuning could reduce any differences, and hybrid algorithms outperform their Grace counterparts at high memory allocations. In our case, this occurs when the ratio of main memory to input size reaches approximately 1:8 (2 M of main memory) or 1:4 (4 M of main memory).

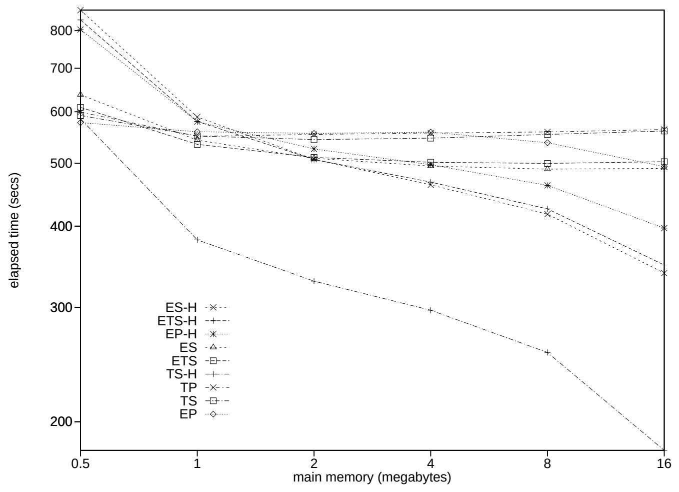

Figure 11: No coalescing (reduction factor = 1, without NL)

We note that TS-H, the temporal sorting hybrid algorithm performed extremely well in this experiment (see Figure 11). The reasons for this are two-fold. First, due to the uniform distribution of both the explicit and timestamp attributes all of the hybrid algorithms incurred essentially the same I/O cost. (The same statement is true for the set of Grace algorithms.) The cost differential between TS-H and the other hybrid algorithms in Figure 11 is therefore main memory cost, in this case, sorting cost. Recall that we are using 4 byte integers as timestamps, and two 4 byte integers as the explicit attribute values. Recall that TS-H (and TS) sort the input using the starting timestamp as the sort key. Hence, the cost of sorting using timestamps is one-half the cost of sorting using the explicit attributes. As tuple length increases this gap would grow correspondingly. The cost differential between TS-H and ETS-H is even higher since ETS-H sorts using both the explicit attribute and the starting timestamp. Lastly, the tuple caching mechanism used in TS-H (and TS) is quite effective in this experiment since the timestamp durations are very short (one chronon). Hence, tuples quickly expire from the tuple cache, with no danger of cache overflow, and subsequent buffer thrashing, is possible. We will revisit this point in Section 6.4.

6.3.2 Total Coalescing

The experiment of the previous section performed essentially no coalescing, due to the uniform distribution of explicit attribute values over a large domain. We now consider the algorithms when the reduction factor approaches the cardinality of the input, i.e., when all tuples coalesce into a one or a few tuples.

A critical factor in the performance of duplicate elimination algorithms is the presence (or absence) of skew in the input attribute values [BD83, Gra93]. Sort-merge algorithms generally out-perform their partition-based counterparts when input distributions are skewed, due to enlarged partition sizes, and subsequent buffer thrashing. For binary operations such as joins, other factors, such as the relative sizes of the input operands, must be considered, but for unary operators such as duplicate elimination or grouping, skew is the primary concern for performance. We investigate explicit attribute skew more closely in the next section, but we mention its effect here to illustrate the results of this experiment.

We generated a single relation consisting entirely of value-equivalent tuples, i.e., every tuple in the relation had the same values for its explicit attributes. (Equivalently, the relations explicit attribute values had 100% skew.) As before the period timestamps of the tuples were randomly distributed with one chronon durations. Notice that 1 M tuples distributed over a 100000 chronon lifespan implies that approximately 10 tuples are valid during each chronon, thereby supporting a large reduction factor. Each algorithm was again run using six memory allocations, from 0.5 M to 16 M. The results are shown in Figure 12. Note that both the x-axis and y-axis are log-scaled.

As expected, EP and EP-H suffer when the explicit attribute values are skewed. ES-H enjoys a 176% advantage over EP-H at the smallest memory allocation, and a 330% advantage at the largest memory allocation. However, notice that EP's performance is slightly improved from its performance in Figure 11. This is due to a sorting optimization, which we applied to all algorithms, but will explain in terms of explicit attribute partitioning. We

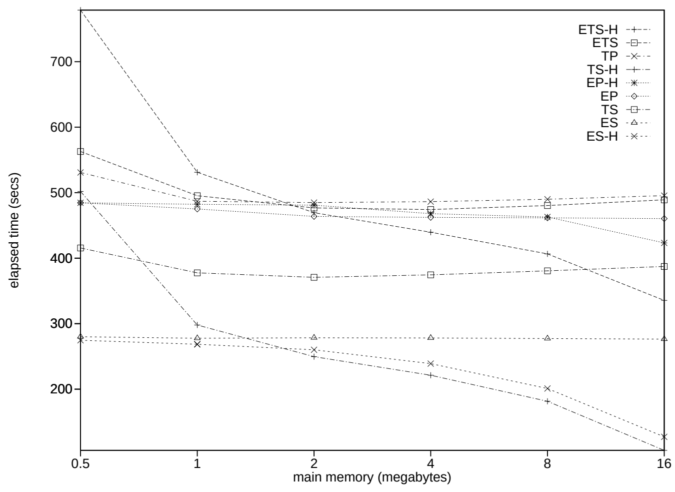

Figure 12: Total coalescing (reduction factor = input cardinality)

used a pointer-based quicksort to coalesce partitions. A partition is read into memory, sorted, and then scanned and coalesced. If a subarray of pointers is found to be already sorted, as would occur when all tuples are value-equivalent, then the algorithm skips the sorting step for that subarray. (The cost of detecting the sortedness of the subarray is included in the algorithm cost.) For this experiment, where all tuples are value-equivalent, this optimization results in a big savings. Nonetheless, the same optimization was applied to ES and ES-H, (and the remaining algorithms as well) so the relative difference between those algorithms and EP and EP-H is the same. This also explains the lack of "tail-up" for ES-H at the smallest memory allocation, as occurs for the other hybrid algorithms.

One further note of interest is the good performance of TS-H at relatively large memory sizes. Again, this is due to the uniform distribution of the timestamp attributes, and their short duration. We will relax this assumption in the next section.

6.4 Explicit Attribute Skew

To further investigate the effect of explicit attribute skew, we generated a series of databases with explicit skew ranging from 0%, where explicit attribute values were randomly distributed as in the experiment of Section 6.3.1, to 100%, as in the experiment of Section 6.3.2. We conducted two experiments. In the first, we used short duration timestamps (one chronon). In both cases, the timestamps were randomly distributed over the timestamp space. To provide as fair a comparison as possible, we fixed the memory size at two megabytes, the size at which all algorithms performed most closely in the experiment of Section 6.3.1. The result of the first experiment is shown in Figure 13. Notice that the y-axis of the graph is log-scaled.

Figure 13 shows no surprises. The relative order of EP/EP-H, ETS/ETS-H, and ES/ES-H reflects the increased I/O cost incurred by EP/EP-H, the higher sorting cost of ETS/ETS-H over ES/ES-H, and the sorting optimization mentioned earlier for ES and ES-H. As expected, TP shows essentially the same cost as in the previous experiment. Again the performance of TS-H is exceptional due to the short duration timestamps.

For the second experiment, we increased the timestamp duration to 1000 chronons, making the lifespan of each tuple 1/100 of the relation lifespan. The results of this experiment are shown in Figure 14.

This experiment shows a serious problem with the timestamp sorting algorithms: both show quadratic cost when the explicit attribute skew is 50% or below. Recall that the only difference between the trial relations in Figures 13 and 14 is the length of the tuple timestamps. The effect of a high explicit attribute cardinality (which is inversely proportional to the degree of explicit skew) coupled with long duration timestamps causes buffer thrashing. Many tuples will require caching due to their long timestamps, but few tuples will coalesce due to the high explicit cardinality. Hence, the cache buffer overflows, and the algorithms begin thrashing.

To see this, recall that our test relation had a lifespan of 100000 chronons, and contained 1 M tuples whose timestamps were randomly distributed throughout the relation lifespan. This implies that approximately 10 = 1 M

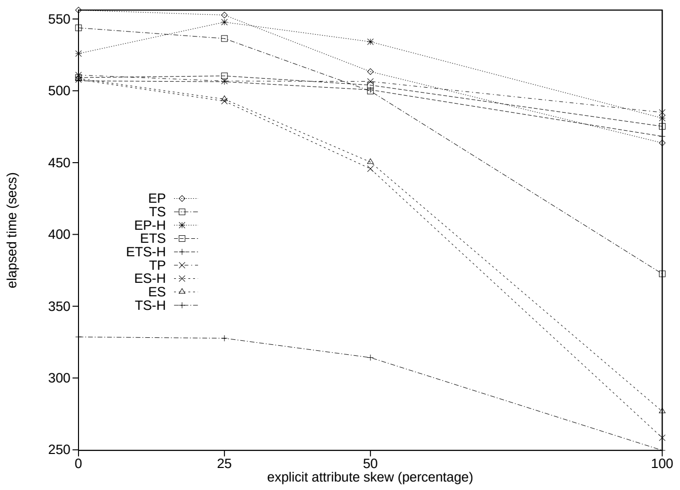

Figure 13: Explicit attribute skew (short duration timestamps)

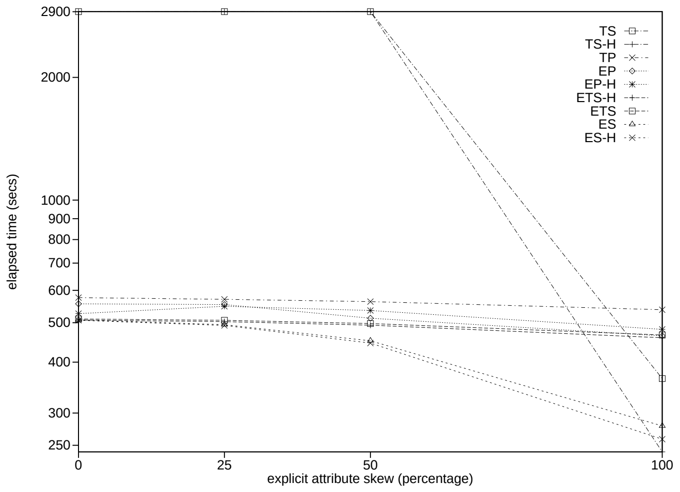

Figure 14: Explicit attribute skew (long duration timestamps)

/ 100000 tuples arrive and depart every chronon. Since each tuple has a lifespan of 1000 chronons, we must scan the first 10000 tuples, approximately, before any tuples can safely be purged from the tuple cache. However, with a 32K cache size (holding 32K / 16 = 2048 tuples), we have already overflowed the tuple cache. Again, this effect was induced by the interplay of a high explicit attribute cardinality and fairly long duration timestamps.

This did not happen in the experiments of Figure 11 and 13 since tuple timestamps there had short durations, i.e., a single chronon. Tuples with short lifespans are less likely to coalesce than tuples with long lifespans. Therefore, short lifespan tuples are more quickly flushed from the tuple cache, thereby reducing the exposure to overflow and subsequent thrashing.

We conjecture that a similar effect occurs when the explicit attribute cardinality is low, but long duration timestamps are still present. Further experimentation is needed to confirm this. Also, notice that the effect would be mitigated if the relation lifespan were longer, thereby decreasing the arrival rate of tuples.

To see the effect of explicit attribute skew on the remaining algorithms, we have replotted their data in Figure 15. From the figure, we can see that EP-H had almost uniformly poorer performance than its Grace counterpart. This is due to the nature of the skew. In none of the hybrid experiments did the skewed partitions happen to be the same as the memory resident partitions, meaning that a large amount of buffer space was left unoccupied during partitioning. More sophisticated buffer allocation techniques could have alleviated this problem.

We make one last observation from this set of experiments. In all pairs of Grace and hybrid variants of the

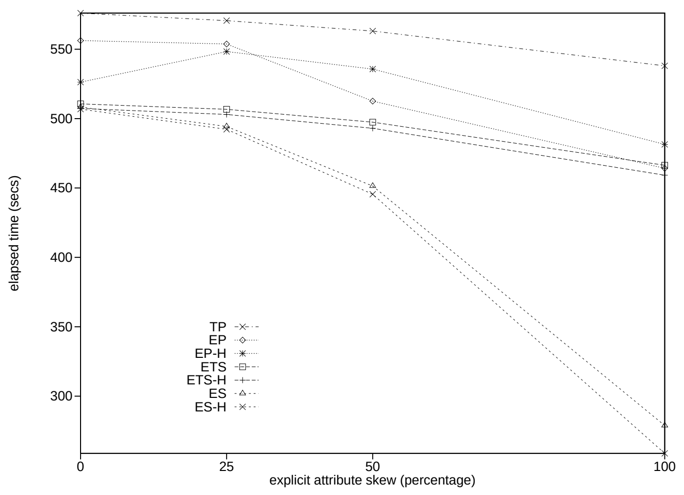

Figure 15: Explicit attribute skew (long duration timestamps, without TS and TS-H)

same algorithms, the evaluation cost shows an overall decrease, rather than increase, as the percentage of skewed tuples rises. This is due to two factors, the sorting optimization mentioned earlier, and the size of the output relation. As the skew increases, more tuples coalesce together, thereby decreasing the size of the output relation.

6.5 Timestamp Attribute Skew

In the previous experiments, we investigated the effects of explicit attribute skew on the coalescing algorithms. We designed an analogous experiment to test the effect of timestamp skew.

We generated databases with increasing timestamp skew, from 0%, where timestamps were randomly distributed, to 100%, where all timestamps had the same values. As before, the explicit attribute values were randomly distributed over the range 0 to $2^{31} - 1$. We used short duration (1 chronon) timestamps to mitigate the effect of long duration timestamps on TS and TS-H. Unfortunately, this was not successful. The relations in this experiment had a high explicit attribute cardinality and increasing timestamp skew. As the timestamp skew rises, more tuples are clustered in time, and must be cached. As the explicit cardinality is high, the likelihood of these tuples coalescing is low. Hence, both TS and TS-H overflowed their caches, and thrashed buffers to disk.

The results of the experiment are shown in Figure 16. As the performance of TS and TS-H closely follows their performance in Figure 14, we omit their lines. As can be seen, the expected result, where TP suffers due to the timestamp skew,is obtained.

6.6 Combined Explicit/Timestamp Attribute Skew

In the previous experiments, we tested the performance of the algorithms under explicit and timestamp skew separately. We now consider the simultaneous effect of both types of skew.

As before, we generated a series of databases with increasing explicit and timestamp skew from 0% (random distributions) to 100% (all tuples are identical). The results are shown in Figure 17.

It is somewhat surprising to see that TS, TS-H, and TP exhibit comparable performance to the other algorithms in Figure 17, given their performance under similar input in the previous experiments. The difference is in the way the input relation has been skewed. We created a single tuple, with a fixed period timestamp and explicit attribute values. Skew, for both the timestamp and explicit attribute dimensions, was created by adding increasing numbers of this tuple to the input relations. In effect, this synchronized the timestamp and explicit skew.

The effect of this procedure on TS and TS-H was to enhance the efficiency of the tuple cache since many value-equivalent tuples with identical timestamps would coalesce, and prevent overflow. For TP, the effect of partitioning the skewed data is offset somewhat by the reduced in-memory sorting cost. This is evidenced by the rise in cost between the 25% and 50% trials, and the subsequent decrease at the 100% trial.

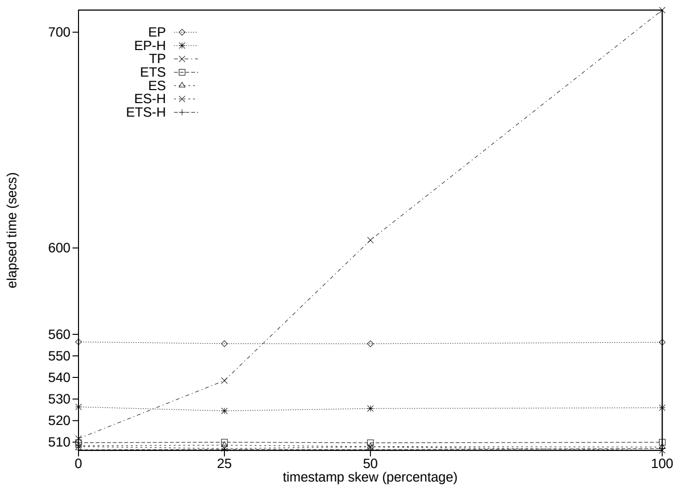

Figure 16: Timestamp skew

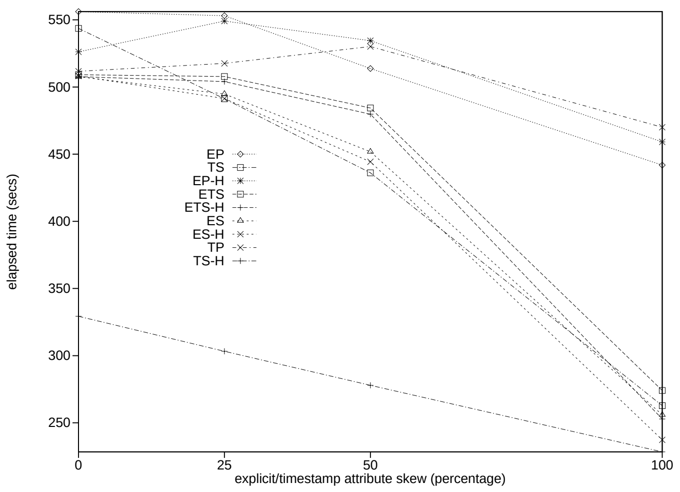

Figure 17: Combined explicit/timestamp skew

6.7 Summary

We investigated the performance of nested-loop (NL), explicit partitioning (EP and EP-H), explicit sorting (ES and ES-H), temporal sorting (TS and TS-H), temporal partitioning (TP), and combined explicit/temporal sorting (ETS, ETS-H) when performing coalescing. The timestamp-based algorithms, especially TS and TS-H, show good performance in special cases, e.g., when timestamps are randomly distributed, are short in duration, and little value-equivalence is present in the explicit attributes. However, the performance of these algorithms degrades quickly when timestamp durations increase, causing tuple caching for TS and TS-H and replication for TP, or when the explicit and timestamp distributions interact in certain ways, e.g., high explicit cardinality with increasing timestamp skew. While these algorithms appear beneficial in certain circumstances, but it would be unwise for a DBMS to rely on a solely on them.

Unlike the timestamp-based algorithms, EP, EP-H, ES, and ES-H, along with the simple variants ETS and ETS-H, show relatively stable performance. As in conventional databases, the performance of EP and EP-H degrade when explicit skew is present. However, ES, ES-H, ETS, ETS-H show relatively stable performance in the presence of explicit skew and timestamp skew, as expected. This is good news for commercial vendors interested in implementing temporal operations—they can construct temporal operators easily by modifying their existing software base, at presumably small development cost, and still achieve very acceptable performance. The choice of between ES/ES-H and ETS/ETS-H is a tradeoff between the additional sorting expense ETS and ETS-H incur to secondarily order the input on time, and the cost that ES and ES-H incur to build in-memory temporal elements. While the space requirement for temporal element construction is usually small, the size of the available buffer space may be the deciding factor. Of course, when sufficient main memory is available, hybrid algorithms should be chosen over their Grace counterparts.

7 Conclusions

Temporal coalescing is an important operation in terms of both database semantics and performance. While many temporal data models and languages have implicitly or explicitly assumed or provided coalescing, only recently has the importance of this operator to performance been emphasized. We hope the work described in this paper will help focus the research community's attention on this operation.

The contributions of this paper can be summarized as follows.

- We motivated the importance, and difficulty, of performing coalescing.
- We defined rules for eliminating superfluous coalescing operations in algebraic expressions.
- We showed how database users can implement coalescing through SQL expressions and investigated the performance of three different ways of doing this. The best performance, which was quadratic in the number of input tuples, was achieved by iterating an SQL update statement. We also showed that the relational algebra has sufficient expressive power to implement coalescing, by exhibiting a (non-iterated) SQL select statement for coalescing.
- We devised seven algorithms which could be used as part of an internal DBMS implementation for coalescing, and compared their performance under a variety of database conditions. The conclusion was that existing algorithms can be augmented to implement coalescing, at presumably modest development cost, and with acceptable performance.

In terms of future research, more work is needed to understand the interplay of coalescing and other temporal operators with respect to query optimization and evaluation. Concerning query optimization, existing approaches, such as predicate pushdown [Ull88] and pullup [HS93, Hel94], early and late aggregation (c.f., [YL94], duplicate elimination removal [PL94], and `DISTINCT` pullup and pushdown, should be applied to coalescing. Effective cost formulas for coalescing are needed. Concerning evaluation, composite operators which implement two or more operators from a base set of algebraic operators could be exploited. Finally, a more thorough study of the algorithm space for temporal coalescing would be beneficial to identify cases where specialized temporal algorithms can be exploited.

8 Acknowledgments

The second and third authors were supported in part by NSF grant ISI-9202244 and a grant from the AT&T Foundation. The authors thank Goetz Graefe for his insights.

References

[All83] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 16(11):832–843, 1983.

[Ari86] G. Ariav. A Temporally Oriented Data Model. *ACM Transactions on Database Systems*, 11(4):499–527, December 1986.

[BCST96] M. Böhlen, J. Chomicki, R. Snodgrass, and D. Toman. Querying TSQL2 Databases with Temporal Logic. In *Proceedings of the International Conference on Extended Database Technology*, to appear, 1996.

[BJS95] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Evaluating and Enhancing the Completeness of TSQL2. Technical Report TR 95-5, Computer Science Department, University of Arizona, 1995.

[Böh94] M. Böhlen. *The Temporal Deductive Database System ChronoLog*. PhD thesis, Departement Informatik, ETH Zürich, 1994.

[BD83] D. Bitton and D. J. DeWitt. Duplicate Record Removal in Large Data Files. ACM Transactions on Database Systems, 8(2), June 1983, p. 255.

[BZ82] J. Ben-Zvi. *The Time Relational Model*. PhD thesis, Computer Science Department, UCLA, 1982.

[CC87] J. Clifford and A. Croker. The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans. In *Proceedings of the International Conference on Data Engineering*, pages 528–537. IEEE Computer Society, IEEE Computer Society Press, February 1987.

[Cel95] J. Celko. *SQL for Smarties: Advanced SQL Programming*. Morgan Kaufmann, 1995.

[DKO$^+$84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–8, June 1984.

[DNS91] D. DeWitt, J. Naughton, and D. Schneider. An Evaluation of Non-Equijoin Algorithms. In *Proceedings of the Conference on Very Large Databases*, p. 443, 1991.

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

[GLS94] G. Graefe, A. Linville, and L. D. Shapiro. Sort vs. hash revisited. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):934–944, December 1994.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[GV85] S. K. Gadia and J. H. Vaishnav. A Query Language for a Homogeneous Temporal Database. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 51–56, March 1985.

[Hel94] J. M. Hellerstein. Practical Predicate Placement. In *Proceedings of the ACM Conference on Management of Data*, pages 325–335, June 1994.

[HS93] J. M. Hellerstein and M. Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proceedings of the ACM Conference on Management of Data*, pages 267–276, May 1993.

[JCE$^+$94] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes, and S. Jajodia. A Glossary of Temporal Database Concepts. *SIGMOD RECORD*, 23(1):52–64, March 1994.

[KS95] N. Kline and M. D. Soo. TIME-IT*: The* TIME *Integrated Testbed*. Pre-beta version 0.1 available via anonymous ftp from `ftp.cs.arizona.edu`, September 1995.

[KSL95] N. Kline, R. T. Snodgrass, and T. Y. C. Leung. Aggregates. In R. T. Snodgrass, editor, *The TSQL2 Temporal Query Language*, chapter 21, pages 395–425. Kluwer Academic Publishers, 1995.

[LJ88] N. Lorentzos and R. Johnson. Extending Relational Algebra to Manipulate Temporal Data. *Information Systems*, 15(3), 1988.

[LM90] C. Leung and R. Muntz. Query Processing for Temporal Databases. In *Proceedings of the International Conference on Data Engineering*, February 1990.

[LM91] T. Y. C. Leung and R. R. Muntz. Temporal Query Processing and Optimization in Multiprocessor Database Machines. Technical Report CSD-910077, Computer Science Department, University of California, Los Angeles, November 1991.

[LM93] T. Y. C. Leung and R. R. Muntz. Stream Processing: Temporal Query Processing and Optimization. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, chapter 14, pages 329–355. Benjamin/Cummings Publishing Company, 1993.

[Lor93] N. Lorentzos. The Interval-extended Relational Model and Its Application to Valid-time Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, chapter 3, pages 67–91. Benjamin/Cummings Publishing Company, 1993.

[LP95] T. Y. C. Leung and H. Pirahesh. Querying Historical Data in IBM DB2 C/S DBMS Using Recursive SQL. In J. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, Workshops in Computing, Zürich, Switzerland, September 1995. Springer Verlag.

[McK88] E. McKenzie. *An Algebraic Language for Query and Update of Temporal Databases*. PhD thesis, University of North Carolina, Computer Science Department, September 1988.

[MS91] L. E. McKenzie and R. T. Snodgrass. Evaluation of Relational Algebras Incorporating the Time Dimension in Databases. *ACM Computing Surveys*, 23(4):501–543, December 1991.

[MS93] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers, Inc., 1993.

[NA89] S. B. Navathe and R. Ahmed. A Temporal Relational Model and a Query Language. *Information Systems*, 49(2):147–175, 1989.

[NA93] S. Navathe and R. Ahmed. Temporal Extensions to the Relational Model and SQL. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 92–109. Benjamin/Cummings Publishing Company, 1993.

[O'N94] P. O'Neil. *Database Principles Programming Performance*. Morgan Kaufmann, San Francisco, 1994.

[PL94] G. N. Paulley and P.-A. Larson. Exploiting Uniqueness in Query Optimization. In *Proceedings of the International Conference on Data Engineering*, pages 68–79, February 1994.

[SA86] R. T. Snodgrass and I. Ahn. Temporal Databases. *IEEE Computer Journal*, 19(9):35–42, September 1986.

[Sar90a] N. Sarda. Algebra and Query Language for A Historical Data Model. *IEEE Computer Journal*, 33, 1990.

[Sar90b] N. Sarda. Extensions to SQL for Historical Databases. *IEEE Transactions on Knowledge and Data Engineering*, pages 220–230, June 1990.

[Seg93] A. Segev. Join Processing and Optimization in Temporal Relational Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, chapter 15, pages 356–387. Benjamin/Cummings Publishing Company, 1993.

[SJS95] M. D. Soo, C. J. Jensen, and R. T. Snodgrass. An Algebra for TSQL2. In R. T. Snodgrass, editor, *The TSQL2 Temporal Query Language*, chapter 27, pages 505–546. Kluwer Academic Publishers, 1995.

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

[Sno95] R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[Sri91] S. M. Sripada. *Temporal Reasoning in Deductive Databases*. PhD thesis, Imperial College of Science and Technology, University of London, 1991.

[SSD87] R. Sadeghi, W. B. Samson, and S. M. Deen. HQL — A Historical Query Language. Technical report, Dundee College of Technology, Dundee, Scotland, September 1987.

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

[Tan86] A. U. Tansel. Adding Time Dimension to Relational Model and Extending Relational Algebra. *Information Systems*, 11(4):343–355, 1986.

[TCG$^+$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.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, Volume I. Computer Science Press, 1988.

[YL94] W. P. Yan and P.-A. Larson. Performing Group-By Before Join In *Proceedings of the International Conference on Data Engineering*, pages 89–100, February 1994.