Coalescing in Temporal Databases

Michael H. Böhlen
Dept. of Math and Computer Science
Aalborg University
Fredrik Bajers Vej 7E
DK-9220 Aalborg Ø, DENMARK
boehlen@iesd.auc.dk

Richard T. Snodgrass
Dept. of Computer Science
University of Arizona
Tucson, AZ 85721
rts@cs.arizona.edu

Michael D. Soo
Dept. of Computer Science and Engr.
University of South Florida
4202 E. Fowler Ave, ENB118
Tampa, FL 33620
soo@csee.usf.edu

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 sketch a variety of iterative and non-iterative approaches, via SQL and embedded SQL, 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, temporal partitioning, and combined explicit/temporal sorting, and summarize a performance study involving a subset of these algorithms. The study shows that coalescing can be implemented with reasonable efficiency, and with modest development cost.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996

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

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 and 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.

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, TCG+93, McK88, SSD87], 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, coalescing the relation prior to evaluating the query causes Ron to qualify. 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 coalesced relations. 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 define the space of algorithms to evaluate coalescing within a DBMS and summarize a performance study involving a subset of these algorithms. 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 [TCG+93, p.202–208], Clifford and Croker's Historical Relational Data Model (HRDM) [TCG+93, p.6–27], Navathe's Temporal Relational Model (TRM) [TCG+93, p.92–109], and the data models defined by Gadia [TCG+93, p.28-66], Sadeghi [SSD87] and Tansel [TCG+93, p.183-201] 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 [TCG+93, p.110-140] 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` [TCG+93, p.125ff]. Some query languages that don't require coalesced relations provide constructs to explicitly specify coalescing; ChronoBase [Sri91], ChronoSQL [Böh94], VT-SQL [Lor93] and ATSQL2 [BJS95] are examples. The SQL/Temporal part of SQL3 [Mel96] contains a `NORMALIZE ON` clause that coalesces on specified columns.

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 [TCG+93, p.108]. Sarda defined an operator called Coalesce [TCG+93, p.122], Lorentzos' FOLD operator includes coalescing [TCG+93, p.75], 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" [TCG+93, p. 337], implying 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 expres-

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

sive power to the relational algebra. In Section 5, we settle the question by proving that coalescing is expressible in the (nonrecursive) 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 [TCG+93, p.108]. Lorentzos uses a similar algorithm to implement FOLD [TCG+93, p. 89]. In Section 6, we evaluate this algorithm against a suite of other approaches.

3 Basic Definitions

While coalescing can be defined in almost all temporal data models, we choose a particular model 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 $t_s \leq t < t_e$. In this paper, we consider *valid-time* relations [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 $R = (A_1, \ldots, A_N \| \mathrm{VT})$ as a set of *explicit attributes* $\{A_1, \ldots, A_N\}$ and a 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.

$$value_eq(x, y) := (x[A] = y[A])$$

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.

$$adj([S_1 - E_1), [S_2 - E_2)) := (E_1 = S_2 \vee S_1 = E_2)$$

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

$$ovlp([S_1 - E_1), [S_2 - E_2)) := (\exists c((S_1 \leq c < E_1) \wedge (S_2 \leq c < E_2)))$$

Finally, a relation instance r of the schema $R = (A_1, \ldots, A_N \| VT)$ is coalesced if all pairs of distinct tuples from the relation are either not value-equivalent, or, if they are value-equivalent, then they must be non-adjacent and non-overlapping.

$$is_coal(r) := \forall x, y \in r(x = y \vee \neg value_eq(x, y) \vee (\neg ovlp(x[VT], y[VT]) \wedge \neg adj(x[VT], y[VT])))$$

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, TCG+93, 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]. To each relational algebra operator *op* a valid time counterpart op^v exists that defines the semantics of valid time [BSS96]. 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 describe 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

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

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.

It is possible to identify operations that potentially destroy coalescing and operations 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 $r_1 = \{\langle a, b \| [2\text{–}5) \rangle, \langle a, c \| [5\text{–}8) \rangle\}$ and $r_2 = \{\langle a, b \| [5\text{–}9) \rangle\}$. We have $\pi^v_A(r_1) = \{\langle a \| [2\text{–}5) \rangle, \langle a \| [5\text{–}8) \rangle\}$ 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. □

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 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.

(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, i.e., if the input is coalesced the output relation is coalesced too.

(r1) $\quad coal(r_1 \times^v r_2) \equiv coal(r_1) \times^v coal(r_2)$
(r2) $\quad coal(r_1 \setminus^v r_2) \equiv coal(r_1) \setminus^v coal(r_2)$
(r3) $\quad coal(\sigma^v_c(coal(r_1))) \equiv \sigma^v_c(coal(r_1))$

Temporal Cartesian product and temporal set difference both preserve coalescing. Moreover, if coalescing is carried out, it does not matter whether coalescing is applied before or after the respective operation. 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.

(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^v_f, 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, which is a vector of expressions reflecting the select list of SQL statements, computes a value based on the valid-time of the input relation).

A final unconditional optimization applies to temporal set difference.

(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. This is easily checked with the syntactic condition $vt_free(c)$ which ensures that the valid time is not used in the condition c.

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

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.

$$\text{(r7)} \quad coal(\pi_f^v(coal(r_1))) \equiv coal(\pi_f^v(r_1)) \text{ iff } vt_free(f)$$

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

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

In data models that enforce coalesced base relations 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 one explicit attribute. Valid-time is represented by two attributes, S, denoting the start point and E, denoting the end point.

To empirically determine upper and lower bound costs for coalescing we have used two instantiations of r, each with n tuples. (See Section 6 for a discussion of different database instances.) The first one, r_1, consists of a single chain of adjacent value-equivalent tuples. 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, contains no value-equivalent tuples, thus, the reduction factor is 1.

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. Searching for value-equivalent tuples with overlapping or adjacent periods has to be done with a self-join.

$$r(X\|[A-B)) \wedge r(X\|[C-D)) \wedge \\ A < C \wedge C \leq B \wedge B < D$$

An optimization exploits the fact that we are only interested in maximal periods. 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 is repeated until r doesn't change anymore. 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).

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 were 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.

$$
\begin{aligned}
& r(X\|[S-_)) \wedge r(X\|[_-E)) \wedge S < E \wedge \\
& \forall A(r(X\|[A-_)) \wedge S < A < E \Rightarrow \\
& \qquad \exists U, V(r(X\|[U-V)) \wedge U < A \leq V)) \wedge \\
& \neg\exists A, B(r(X\|[A-B)) \wedge (A < S \leq B \vee A \leq E < B))
\end{aligned}
$$

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 and third line ensure 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 $\forall$-quantifiers and implications [BCST96] and then translating to SQL directly [BSS96]. The resulting select statement is considerably more complex than the insert or update statement discussed in the previous section. 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, e.g., possible to create an index on the valid time start point for the purpose of coalescing. The effectiveness of such optimizations 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 2 summarizes the performance of the three approaches, along with the update algorithm using an index. All were performed on the uncoalesced relation r_1. 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.

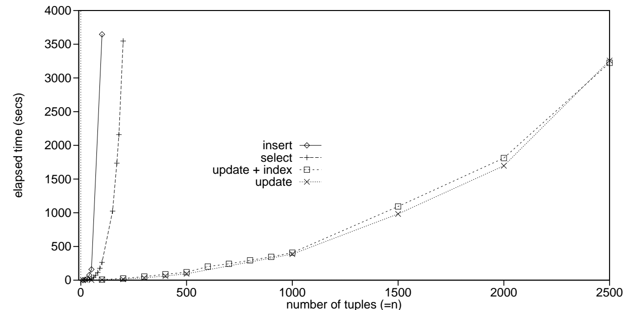

Figure 2: Coalescing a relation where all tuples can be coalesced into a single one.

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

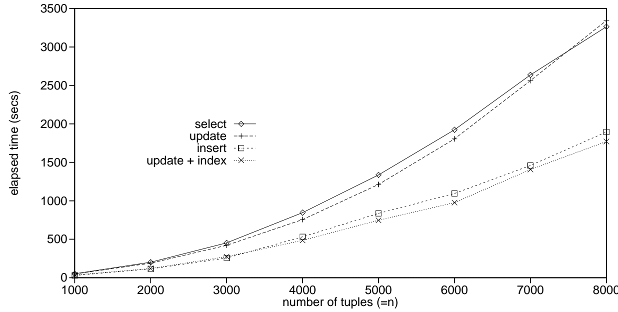

Figure 3: Coalescing a relation without value-equivalent tuples.

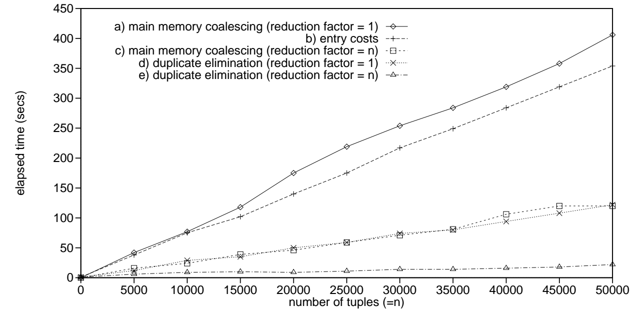

Figure 4: Coalescing a relation by loading it into main memory and then storing it back, as compared with the entry costs and duplicate elimination.

cluded these measurements because of the really 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 C code is given elsewhere [BSS96].

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 4 illustrates the costs to coalesce the template relations. 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 the previous section.) The reason is that fewer 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 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 briefly summarize 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 Algorithm Space

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 [TCG+93, p.329–387]. 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 [TCG+93, p.92–109]. 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 [TCG+93, p.329–387]. 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 5 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 6.

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

Figure 5: Space of possible evaluation algorithms

6.2 Algorithm Implementations

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 [LM92] 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

Algorithm	Name
Explicit sort	ES
Hybrid explicit sort	ES-H
Temporal sort	TS
Hybrid temporal sort	TS-H
Explicit/temporal sort	ETS
Hybrid explicit/temporal sort	ETS-H
Temporal/explicit sort	TES
Hybrid temporal/explicit sort	TES-H
Explicit partitioning	EP
Hybrid explicit partitioning	EP-H
Temporal partitioning	TP
Hybrid temporal partitioning	TP-H
Explicit/temporal partitioning	ETP
Hybrid explicit/temporal partitioning	ETP-H
Temporal/explicit partitioning	TEP
Hybrid temporal/explicit partitioning	TEP-H
Nested-loop	NL

Figure 6: Possible algorithms for performance study

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, EP, TP, EP-H, and NL we build the temporal element [TCG+93, p.34] of value-equivalent tuples as a main-memory data structure. The temporal element is represented as a binary tree, where nodes in the tree contain a time period, whose start time is the tree's sort key, and left and right child pointers. On insertion, a new period is coalesced into an existing node if it is possible to do so without violating the sortedness of the tree, otherwise a new leaf node is added. As this scheme may not eliminate all overlapping or adjacent time periods, maximal intervals are produced via a final tree traversal performed after all value-equivalent tuples have been scanned. In all cases, the space requirement of the temporal element was small relative to the available buffer space. Algorithms that employ sorting on time, i.e., TS/TS-H and ETS/ETS-H, do not require this data structure, as the sort ordering on time ensures that each set of value-equivalent tuples can be coalesced within a constant in-memory workspace.

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 7: System characteristics

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 KB, i.e., one cluster of I/O (see Figure 7). 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.

6.3 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 7. (The page size of 1 KB 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 8. 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

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 8: Cost metrics

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.

6.4 Performance Summary

We implemented the ten algorithms NL, EP, EP-H, ES, ES-H, TS, TS-H, TP, ETS, and ETS-H described above, and evaluated their performance under a variety of database and system parameters. We summarize the results of this study here—full details are provided elsewhere [BSS96].

Nested-loop (NL) was not competitive under any circumstances. Like some of the SQL solutions of Section 5, the nested-loop program effectively performed a fix-point 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 essentially the approach proposed by Navathe and Ahmed for the COMPRESS operator [TCG+93, p.92–109] and by Lorentzos for his Fold operator [TCG+93, p.67–91]. 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, and had uniformly better performance than NL at all memory allocations.

The timestamp-based algorithms, especially TS and TS-H, showed good performance in special cases, e.g., when timestamps are randomly distributed and are short in duration, and when little value-equivalence is present in the explicit attributes. However, the performance of these algorithms degraded quickly when timestamp durations increase, due to tuple caching and tuple replication, or when the explicit and timestamp distributions interact in certain ways, e.g., a high explicit cardinality with increasing timestamp skew. While these algorithms appear beneficial in certain circumstances, it would be unwise for a DBMS to rely 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 in SQL and investigated the performance of three different ways of doing this. We showed that the relational algebra has sufficient expressive power to implement coalescing. Database users get the best performance if they fetch tuples ordered first on explicit attributes and second on the temporal attribute. A single tuple is kept in memory and whenever a new tuple is fetched the memory tuple is either updated or stored back and replaced.
- We proposed seventeen algorithms which could be used as part of an internal DBMS implementation for coalescing, and compared the performance of ten of these algorithms. 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 and the anonymous reviewers for their insights.

References

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

[Ari86] G. Ariav. A Temporally Oriented Data Model. *ACM TODS*, 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*, France, March, 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.

[BSS96] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. R-96–2026, Aalborg University, Department of Mathematics and Computer Science, Denmark, June 1996.

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

[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*, pp. 1–8, June 1984.

[DNS91] D. DeWitt, J. Naughton, and D. Schneider. An Evaluation of Non-Equijoin Algorithms. In *Proceedings of the International Conference on Very Large Data Bases*, p. 443–452, 1991.

[Gad88] S. K. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM TODS*, 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.

[Hel94] J. M. Hellerstein. Practical Predicate Placement. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, R. Snodgrass and M. Winslett, editors, pp. 325–335, June 1994.

[HS93] J. M. Hellerstein and M. Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proceedings of the ACM SIGMOD Internation Conference on Management of Data*, San Jose, CA, M. Carey and D. Schneider, editors, pp. 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. *ACM 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 Lan-*

guage, chapter 21, pp. 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.

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

[Lor93] N. Lorentzos. Specification of Valid Time SQL. ESPRIT III Project 7224 (ORES) Deliverable D2, April, 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.

[Mel96] Melton, J. (ed.) *SQL/Temporal.* ISO/IEC JTC 1/SC 21/WG 3 DBL-MCI-009. March, 1996.

[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.

[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*, Houston, TX, pp. 68–79, February 1994.

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

[Sno87] R. T. Snodgrass. The Temporal Query Language TQuel. *ACM TODS*, 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*, Houston, TX, pp. 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*, Houston, TX, pp. 89–100, February 1994.