Optimizing Temporal Queries: Efficient Handling of Duplicates

Ivan T. Bowman and David Toman
Department of Computer Science, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
E-mail: {itbowman,david}@uwaterloo.ca

Abstract

Recent research in the area of temporal databases has proposed a number of query languages that vary in their expressive power and the semantics they provide to users. These query languages represent a spectrum of solutions to the tension between clean semantics and efficient evaluation. Often, these query languages are implemented by translating temporal queries into standard relational queries. However, the compiled queries are often quite cumbersome and expensive to execute even using state-of-the-art relational products. This paper presents an optimization technique that produces more efficient translated SQL queries by taking into account the properties of the encoding used for temporal attributes. For concreteness, this translation technique is presented in the context of SQL/TP; however, these techniques are also applicable to other temporal query languages.

1 Introduction

The last decade of research in the area of temporal databases has led to the development of several temporal query languages based on extensions of existing relational languages, in particular of SQL [7, 9, 10, 11, 12]. These languages are based on the idea of *timestamping* tuples: associating them with a time instant at which the tuple is valid. These instants are usually drawn from a linearly-ordered universe that models time in the database. The semantics of queries are then defined in terms of the individual time instants [3].

Although query semantics is defined in terms of time instants, the space requirements of explicitly storing tuples with individual instants is prohibitive: tuples would need to be repeated for each instant at which they are valid. Instead, practical languages rely on compact encodings of sets of time instants (often referred to as periods of validity), for example by using intervals, *bitemporal elements*, or other interval-based encodings. The chosen encoding is then used as the *concrete* domain for the representation of values of temporal attributes.

Therefore, independently of the query language used, the queries are ultimately evaluated over such a encoding. The query evaluation is often based on *translating* the original query to a standard relational query that can be executed over the encoding [12] or [9, 10, 11] (for snapshot and sequenced fragments). However such compilation based approaches often lead to very complex queries that are extremely difficult to optimize (given current state-of-the-art query optimizers).

The main obstacle here is the fact that the SQL query optimizer has no knowledge of the properties of the *encoding* used for temporal attributes and therefore cannot take advantage of rewrites valid under the semantics of the original temporal queries. This shortcoming stands out especially when following the strict duplicate-preserving semantics of SQL. Consider the following example:

Example 1 Let $D_1 = \langle R : \{(a, [0, 10]), (a, [11, \infty])\}\rangle$ and $D_2 = \langle R : \{(a, [0, \infty])\}\rangle$ be two (semantically equivalent) instances of a temporal database (both asserting that the value a is in the relation R at all time instants $t \geq 0$).

Now consider the query "give me all the values in the first attribute of R". Under the *set* semantics, the answer for both D_1 and D_2 is a single tuple containing the value "a". However, answering this query under a duplicate-preserving semantics is much more complicated: we would have to produce a tuple a *for every time instant* associated with a—this is clearly not possible as the result would have to be infinite.

For this reason, SQL/TP (cf. Section 2 or [12, 13] for description of the language) *prohibits* duplicate-preserving projections of temporal attributes.

Example 2 The commonly proposed solution for implementing the query from Example 1 that simply projects out the interval attribute [10] (nonsequenced fragment) is also problematic: the result of the above query differs for D_1 and D_2 in the number of duplicate "a"'s returned (2 tuples for D_1 and 1 for D_2). It has been argued however, that no well-behaved temporal query should distinguish between equivalent temporal databases. While one might argue that the difference between the instances D_1 and D_2 can be resolved using coalescing [2], it has been shown that, in general, coalescing-based approaches are bound to fail [4, 12].

Example 3 Consider the following example. A software development company maintains a time reporting database that records the projects that each developer worked on throughout the year. The relation used for this reporting has the following signature:

```
Assignment(Date,Employee,Project).
```

Table 1 contains a concrete instance of this relation.

Date	Employee	Project
Jan – July	Bob	A
July – Dec	Bob	B
Jan – Mar	Mary	A
Mar – Dec	Mary	A
Nov – Dec	Mary	B

Table 1. Time-Reporting Example Database

If we are interested in the time reporting for only one employee, say Mary, we can write the query Q in SQL/TP as follows:

```
SELECT DISTINCT Month, Project
FROM   Assignment
WHERE  Employee = 'Mary'
```

We can use query Q to build up larger queries. If we are interested only in knowing the projects that Mary worked on, we can write query Q_2 as:

```
SELECT DISTINCT Project
FROM   Q
```

If the query translator is aware that Q is being evaluated in a context where duplicates don't affect the result (for example, because of the `DISTINCT` keyword in query Q_2), then the translator could simply project out the temporal attribute of Q. This generates duplicate tuples, which are then eliminated by the `DISTINCT` of query Q_2.

On the other hand, if we are interested in which months Mary has reported working, we could write the query Q_3:

```
SELECT DISTINCT Month
FROM   Q
```

In this case, we need a more sophisticated technique (implemented by *coalescing* in some languages and as *normalization* in SQL/TP to eliminate duplicates within the range-encoded time intervals in order to calculate Q.

In this example, the context of evaluation of a query can be used to choose a more efficient evaluation technique.

The above examples set up the scene for this paper: while we want to use a well-behaved, declarative language such as SQL/TP, we would also like to employ the simple *temporal projection* that projects out the *compact representation* of sets of time instants (as in Example 2) and thus provides a vastly better performance (similarly to duplicate preserving vs. distinct projections in SQL).

The contributions of the paper are twofold:

1. First, we identify contexts in a given query in which relaxed SQL/TP–to–SQL/92 translation rules can be used. In addition we provide these additional compilation rules and show their effect on several examples.
2. Second, and more importantly, the proposed approach provides a general paradigm for optimizing temporal queries (and in general, queries over non-trivial encodings of data) that leads beyond the optimizations possible in standard SQL.

We demonstrate the approach on SQL/TP, the query language proposed in [12, 13]. However, we would like to stress that the proposed approach is applicable to other proposals, e.g., to IXRM [7] of the SQL/Temporal–TSQL2–ATSQL2 family [9, 10, 11].

The remainder of this paper is organized as follows. Section 2 provides an introduction to the SQL/TP query language (more detailed description can be found in [12, 13]). Section 3 describes the main results of the paper: it defines the *duplicate insensitive* evaluation contexts. Section 4 shows patterns where these contexts can simplify the SQL/TP–to–SQL translation. Section 4.1 outlines the modified compilation procedure from SQL/TP to standard SQL. Finally, Section 5 presents our conclusions and discusses several open questions and directions for future research.

2 SQL/TP Primer

The SQL/TP language operates on a *point-based* view of time. Temporal attributes are drawn from a discrete, countably infinite linearly ordered set without endpoints. In addition to time, we also use all of the standard data types provided by SQL such as strings, integers, floats, and so on. Since we do not assign any a-priori meaning to these data types, we refer to them as the *uninterpreted constants*.

```
<query> ::= <rid>
          | SELECT [<sexp> AS <id> {, <sexp> AS <id> }] <query>
          | FROM <query> <rid> {, <query> <rid> }
          | <query> WHERE <condition>
          | <query> GROUP BY <id> {, <id>} [AGG <cexp> as <id> {, <cexp> as <id>}]
          | <query> <setop> <query>

<setop> ::= UNION [ALL] | EXCEPT [ALL | IN] | INTERSECT [ALL | IN]
```

Figure 1. Syntax of SQL/TP

The relationship between the time instants and the uninterpreted constants is captured in a finite set of temporal relations in a database. We distinguish between the *abstract* temporal database, which is defined in terms of time points, and the *concrete* temporal database, which is a compact encoding of the abstract database.

Definition 4 (Abstract Temporal Database) A *signature* sig(R) of a relational symbol R is a tuple $(a_1 : t_1, \ldots, a_k : t_k)$ where a_i are distinct attribute names, t_i are the corresponding attribute types, and k is the *arity* of R. Attributes of type `time` are *temporal attributes*, and the others are *data attributes*.

A *database schema* is a finite set of relational symbols $R_1, \ldots, R_n$ paired with signatures sig$(R_1), \ldots,$ sig(R_n). A *table* R is a (possibly infinite) bag of tuples that match the signature of R defined in the database schema. An *abstract temporal database* is a set of tables defined by a database schema.

In general, we do not restrict the cardinality of temporal tables: we allow infinite tables as well. However, we do require that the multiplicity of each distinct tuple is finite. Note, too, that there is no restriction on the number of temporal attributes in a relation.

The abstract temporal database provides a natural data model for modeling and querying temporal data based on timestamps. However, the storage cost for a naive representation of such a database would be prohibitively large. In the case of infinite tables, it is not even possible to store the abstract temporal database. To address this issue, SQL/TP uses a *compact encoding* of sets of time instants. The choice of intervals as the compact encoding defines a class of *concrete temporal databases*:

Definition 5 (Concrete Temporal Database) Let R be a relational symbol with signature E. A *concrete signature* corresponding to E is defined as a tuple $\overline{E}$ of attributes that contains (1) A for every data attribute a in E, and (2) I_t for every temporal attribute t in E. The attribute I_t represents the interval encoding of consecutive time instants. A *concrete temporal database schema* is a set of relational symbols and their concrete signatures derived from their signatures in the abstract database schema. A *concrete temporal database* is a set of finite relations defined by a concrete database schema.

To capture the relationship between the abstract and concrete temporal databases, we define a *semantic map* operator $[\![.]\!]$. The meaning of a single concrete tuple $x = (I_t, A_1, \ldots, A_k)$ is a bag of tuples $[\![x]\!] = \{(t, a_1, \ldots, a_k) \mid t \in I_t\}$. Similarly, for concrete tuples with multiple abstract temporal attributes we define the result of the $[\![.]\!]$ operator to be the hypercube resulting from allowing each temporal attribute t_i to range within its interval bounds defined in the concrete tuple.

2.1 Syntax and Semantics

The syntax of the SQL/TP query language is defined by the grammar in Figure 1, which has been slightly simplified to omit details such as the precedence of query constructs. Here, `<cexp>` denotes a column expression (an aggregate function), `<sexp>` denotes a scalar expression, `<condition>` represents an atomic condition, and `<setop>` is a multiset operation. The meaning of $Q(D)$ is defined by induction as follows:

1. $R(D)$ denotes the relation stored in D named R.
2. $(\texttt{FROM}\, Q_1 r_1, \ldots Q_k r_k)(D) = Q_1(D) \times \ldots \times Q_k(D)$

   implements Cartesian product.
3. $(Q \,\texttt{WHERE}\, \varphi) = \sigma_\varphi Q(D)$

   is the restriction operation, where φ is an atomic condition (free of boolean connectives). For attributes of type time, φ is one of $a_i + c \leq a_j$, $c \leq a_i$, or $a_i \leq c$, where a_i and a_j are endpoints of temporal attributes and c is a constant.
4. $(\texttt{SELECT}\, e_1 \,\texttt{AS}\, i_1, \ldots, e_k \,\texttt{AS}\, i_k \; Q)(D)$

   is the duplicate-preserving projection operation that generates a bag of tuples with signature $i_1, \ldots, i_k$ and one tuple $(e_1(x), \ldots e_k(x))$ for each tuple $x \in Q(D)$.

5. $(Q$ GROUP BY $a_1, \ldots, a_k$
   AGG $f^1(b_1)$ AS $i_1, \ldots, f^n(b_n)$ AS $i_n)(D)$

   is the grouping operator, where a_i and b_i are attribute names, and f^i are aggregate functions. For temporal attributes, the aggregate functions may be MIN, MAX, or COUNT.

6. $(Q_1$ <SETOP> $Q_2)(D) = Q_1(D)$ op $Q_2(D)$

   are the set (bag) operators. The allowed operations are:

   UNION (set union),
   UNION ALL (additive union),
   EXCEPT (set difference),
   EXCEPT IN ("not exists"),
   EXCEPT ALL (monus),
   INTERSECT (set intersection),
   INTERSECT IN ("exists"), and
   INTERSECT ALL (duplicate-preserving intersection).

Compared to the original definition of SQL/TP [10], we introduce two additional set operations EXCEPT IN and INTERSECT IN. These two operations are the counterparts of SQL's NOT EXISTS and EXISTS constructs in the WHERE clause and their meaning is defined as follows:

$$(Q_1 \ \texttt{EXCEPT IN}\ Q_2)(D) = \{\!| x \in Q_1(D) : x \notin Q_2(D) |\!\}$$
$$(Q_1 \ \texttt{INTERSECT IN}\ Q_2)(D) = \{\!| x \in Q_1(D) : x \in Q_2(D) |\!\}$$

where $\{\!| x \in \ldots |\!\}$ denotes the formation of a bag of x values. Note that from the expressive power point of view, the new operations do not add new capabilities to SQL/TP (as both of them can be simulated by EXCEPT ALL and INTERSECT ALL with the help of additional joins). However, they allow us to extend the inference technique described in Section 3 to a larger class of queries. Further, these operations allow us to express a query that is semantically equivalent to an SQL/TP query but which has a more efficient implementation.

As in the standard SQL/TP, we guarantee closure of all of the above constructs by requiring that the SELECT clause may not project out a temporal attribute encoded using intervals (as this might violate the requirement of finite duplication) and the grouped-by and non-grouped attributes in the GROUP BY—AGG clause are mutually independent whenever aggregate functions COUNT and SUM are used. In addition, to guarantee that the result of a query is representable using the interval encoding we require all attributes in the answer to a query to be pairwise independent.

2.2 Query Compilation

Our final goal is to translate SQL/TP queries to standard SQL and then submit the result to an off-the-shelf database engine. To achieve this goal we have to resolve two problems. First, every temporal attribute t in an SQL/TP query that refers to individual time instants in T has to be replaced with an attribute I_t ranging over intervals, the elements of the concrete temporal sort[1].

Moreover, while the top-level attributes of an SQL/TP query have to be independent, this restriction may not hold for subqueries of the original query. To translate (sub-)queries with dependent attributes we use *conditional queries* of the form $Q\{\varphi\}$ where Q is an SQL translation of the original query (without any references to individual time instants) and φ is a formula that captures the remaining attribute dependencies. The translation itself is then realized by a function comp that maps SQL/TP queries to sets of conditional queries by induction on their structure, while maintaining the following invariant:

$$Q([\![D]\!]) = [\![\mathsf{comp}(Q)(D)]\!] \triangleq \biguplus_{Q^j\{\varphi^j\} \in \mathsf{comp}(Q)} \sigma_{\varphi^j} [\![Q^j(D)]\!]$$

The SQL/TP language carefully models the set and the multiset semantics of SQL—SQL/TP expressions are translated to generate exact multiplicities for all tuples as defined by standard SQL semantics. The compilation procedure thus proceeds inductively on the structure of the original query Q by replacing subqueries by "equivalent" subqueries according to the above invariant [10].

3 Duplicate Insensitive Contexts

For many queries, the requirement to match the precise multiplicity semantics imposes a significant cost in the translated query. For some sub-expressions, this additional cost can be eliminated because the translated expression is evaluated in a context that does not require precise multiplicity semantics. More formally:

Definition 6 (Query Equivalence) Let Q_1 and Q_2 be SQL/TP queries. We say that Q_1 and Q_2 are:

- *set equivalent*, $Q_1 \overset{\text{SET}}{=} Q_2$ if $\{x : x \in Q_1(D)\} = \{x : x \in Q_2(D)\}$ for all databases D;
- *bag equivalent*, $Q_1 \overset{\text{BAG}}{=} Q_2$, if $Q_1(D) = Q_2(D)$ for all databases D, where $=$ is used to denote bag equality unless otherwise noted.

If two queries are set equivalent, they produce the same set of distinct tuples for any given instance. However, the multiplicity of the tuples does not need to agree in the results of the two queries. If, however, two queries are bag equivalent, they produce the same bag of tuples for any given instance.

[1] We assume the existence of an interval-valued data type in the target language; in the absence of such a data type we use pairs of attributes to denote left and right endpoints of intervals instead.

Definition 7 (Context) An expression $C[\,]$ is a *context (of signature σ)* if, for any SQL/TP query Q with signature σ, $C[Q]$ (denoting the syntactical substitution of Q for $[\,]$) is a valid SQL/TP query. We also say that the above queries Q are *compatible* with $C[\,]$.

For example,

$$C[\,] = ([\,]\, \texttt{WHERE}\, \varphi) \text{ and}$$
$$C[\,] = (\texttt{FROM}\, Q_1\ r_1, \ldots, [\,]\ r_i, \ldots, Q_k\ r_k$$

are contexts. For simplicity we omit the context signatures whenever possible.

Definition 8 (Duplicate Insensitive Context) A context $C[\,]$ is *duplicate insensitive* if

$$Q_1 \stackrel{\text{SET}}{=} Q_2 \Longrightarrow C[Q_1] \stackrel{\text{BAG}}{=} C[Q_2]$$

for any pair of queries Q_1, Q_2 compatible with $C[\,]$.

Example 9 Consider the context

$$C[\,] = ([\,]\, \texttt{GROUP BY}\, A).$$

This context is duplicate insensitive since the result of the expression contains one tuple in its output for each distinct value A for any pair of Q_1 set-equivalent to Q_2 (both with the same signature containing the attribute A).

Lemma 10 It is undecidable whether or not an arbitrary context $C[\,]$ is duplicate insensitive.

Proof: Consider the context $C[\,] = (\texttt{FROM}[\,]\ r, Q\ s)$ where Q is a closed query[2]. Then $C[\,]$ is duplicate insensitive if and only if Q is unsatisfiable (empty). However, emptiness is not decidable [1], so there can not exist a decision algorithm for duplicate insensitivity.

3.1 Inference of Duplicate Insensitivity

Although there is no decision algorithm to find precisely whether a given context is duplicate insensitive, we can define syntax directed rules for finding several useful classes of duplicate insensitive contexts.

Lemma 11 (Inference of Duplicate Insensitivity)

1. The contexts

   $[\,]\,\texttt{GROUP BY}\, a_1, \ldots, a_k$
   $\quad \texttt{AGG}\, f^1\, \texttt{AS}\, i^1, \ldots, f^m\, \texttt{AS}\, i^m$,
   $[\,]\,\texttt{INTERSECT}\, Q$, $[\,]\,\texttt{EXCEPT}\, Q$, $[\,]\,\texttt{UNION}\, Q$,
   $Q\,\texttt{INTERSECT}\,[\,]$, $Q\,\texttt{EXCEPT}\,[\,]$, $Q\,\texttt{UNION}\,[\,]$,
   $Q\,\texttt{INTERSECT IN}\,[\,]$, and $Q\,\texttt{EXCEPT IN}\,[\,]$

   are duplicate insensitive. Note that if the `GROUP BY` clause above contains any aggregate functions, they must consist of only the `MIN` and `MAX` aggregate functions; without any aggregates, the expression has the effect of SQL's `SELECT DISTINCT` construct.

2. If $C[\,]$ is a duplicate-insensitive context, then the contexts

   $C[\texttt{FROM}\, Q_1\ r_1, \ldots, [\,]\ r_i, \ldots, Q_n\ r_n]$,
   $C[\texttt{SELECT}\, e_1\, \texttt{AS}\, i_1, \ldots, e_n\, \texttt{AS}\, i_n[\,]]$,
   $C[[\,]\,\texttt{WHERE}\,\varphi]$,
   $C[Q\,\texttt{INTERSECT ALL}\,[\,]]$,
   $C[[\,]\,\texttt{INTERSECT ALL}\, Q]$,
   $C[[\,]\,\texttt{INTERSECT IN}\, Q]$,
   $C[[\,]\,\texttt{EXCEPT IN}\, Q]$,
   $C[Q\,\texttt{UNION ALL}\,[\,]]$, and
   $C[[\,]\,\texttt{UNION ALL}\, Q]$

   where $C[C'[\,]]$ stands for composition of contexts, are also duplicate insensitive.

Proof: The correctness of these inference rules can be demonstrated using a simple case analysis, which is omitted.

4 Duplicate Insensitive Compilation

In this section we turn to the main goal of the paper: we show how the information about duplicate insensitivity of query contexts can be used to improve the SQL/TP compilation to SQL and in turn to generate more efficient SQL queries. We first focus our attention on SQL/TP expressions of the form:

$$Q\, \texttt{GROUP BY}\, a_1, \ldots, a_k, t_1, \ldots, t_l$$

where the a_i are data attributes and the t_i are temporal attributes. This expression corresponds to SQL's duplicate elimination (`SELECT DISTINCT`) construct. Also, this is the *only* way in SQL/TP (modulo syntactic sugar) of projecting out unnecessary temporal attributes, due to the problems shown in Example 1. The standard SQL/TP translation algorithm [12, 13] translates expressions of this form to an SQL query

$$\mathsf{N}^{t_1,\ldots,t_l}_{a_1,\ldots,a_k}[Q;Q]\, \texttt{GROUP BY}\, A_1, \ldots, A_k, T_1, \ldots, T_l,$$

where $\mathsf{N}^{t_1,\ldots,t_l}_{a_1,\ldots,a_k}[Q;Q]$ is a *normalization* operator that guarantees that the interval representation for $t_1, \ldots, t_l$ in any two tuples is either identical or represents disjoint sets of time instants. This translation correctly preserves the meaning of the original query. It is also easy to see that we cannot

[2] Unlike the SQL standard, SQL/TP allows closed queries with the expected semantics.

use optimization based on rewrites purely within SQL/TP and therefore (modified) techniques for "distinct" elimination used by standard SQL optimizers, e.g., [5, 6, 8], cannot be used. In particular the (invalid) SQL/TP expression

$$\texttt{SELECT}\, a_1 \,\texttt{AS}\, a_1, \ldots, a_k \,\texttt{AS}\, a_k, t_1 \,\texttt{AS}\, t_1, \ldots, t_l \,\texttt{AS}\, t_l\ Q$$

does not achieve our goal. Moreover, there is no SQL/TP expression that could replace the original one in a duplicate-insensitive context that would simplify the result of the translation. However, consider *modifying* the SQL/TP–to–SQL translation instead: in duplicate-insensitive contexts we can use the following translation:

$$\begin{aligned}\mathsf{comp}:\quad & Q\,\texttt{GROUP BY}\, x_1, \ldots, x_k \mapsto \\ & \{\texttt{SELECT}\, X_1, \ldots, X_k \,\texttt{FROM}\, Q^j\{\pi_{x_1,\ldots,x_k}\varphi^j\} : \\ & \qquad Q^j\{\varphi^j\} \in \mathsf{comp}(Q)\}\end{aligned}$$

Lemma 12 Let $C[\ ]$ be a duplicate insensitive context. Then

$$C[Q\,\texttt{GROUP BY}\, x_1, \ldots, x_k]([\![D]\!]) \overset{\text{BAG}}{=} [\![\mathsf{comp}(C[R_i])(D)]\!]$$

where the views

$$R_i \in \{\texttt{SELECT}\, X_1, \ldots, X_k \,\texttt{FROM}\, Q^j\{\pi_{x_1,\ldots,x_k}\varphi^j\} : Q^j\{\varphi^j\} \in \mathsf{comp}(Q)\ \}$$

are conditional queries resulting from compilation of C's argument.

Proof: The additive union of the views R_i generates a result that is set equivalent to the original expression. These views are non-uniform, in the sense that the bag results of individual views may depend on the concrete representation of the database. However, the duplicate-insensitive context C guarantees the bag equivalence of the result of the context for any set-equivalent rewriting of the input.

Similarly to handling the projection operator, we can apply this technique to several of the set operations. In particular, we can apply the technique to converting the `INTERSECT`, `INTERSECT IN`, and `INTERSECT ALL` operations into joins and projections and the `EXCEPT` and `EXCEPT IN` to anti-joins within duplicate insensitive contexts.

4.1 Enhancement of Query Compilation

As the last step of the development we incorporate the notion of duplicate insensitive contexts and the simplified translation rule(s) into the SQL/TP–to–SQL/92 compilation. We use two mutually-recursive functions comp^S and comp^B that are used in the duplicate insensitive (set) and duplicate sensitive (bag) contexts, respectively. We follow a convention that if Q^j and φ^j appear in the translation, then they range over all j such that $Q^j\{\varphi^j\} \in \mathsf{comp}^*(Q)$.

Unchanged translation rules [12, 13]:

$$\begin{aligned}&\mathsf{comp}^*(R) = \\ &\quad \{R\{true\}\}\end{aligned}$$

$$\begin{aligned}&\mathsf{comp}^*(\texttt{FROM}\, Q_1\ R_1, \ldots, Q_k\ R_k) = \\ &\quad \{\texttt{FROM}\, Q_1^{j_1}\ R_1, \ldots, Q_k^{j_k}\ R_k\{\varphi_1^{j_1} \wedge \ldots \wedge \varphi_k^{j_k}\}\}\end{aligned}$$

$$\begin{aligned}&\mathsf{comp}^*(Q\,\texttt{WHERE}\, t_i \le t_j) = \\ &\quad \{Q^j\,\texttt{WHERE}\,\Psi(\varphi^j \wedge t_i \le t_j)\{\varphi^j \wedge (t_i \le t_j)\}\}\end{aligned}$$

$$\begin{aligned}&\mathsf{comp}^*(\texttt{SELECT}\, e_1 \,\texttt{AS}\, i_1, \ldots, e_k \,\texttt{AS}\, i_k\ Q) = \\ &\quad \{\texttt{SELECT}\, e'_i \,\texttt{AS}\, i_1, \ldots, e'_k \,\texttt{AS}\, i_k\ Q^j\{\psi^j\}\}\end{aligned}$$

$$\begin{aligned}&\mathsf{comp}^*(Q\ \texttt{EXCEPT ALL}\ R) = \\ &\quad \{Q^i\{\varphi^i \wedge \neg \textstyle\bigvee_{j \in J} \psi^j\}, \\ &\quad\ Q'^i\ \texttt{EXCEPT ALL}\, R'^j\{\varphi^i \wedge \psi^j\}\}\end{aligned}$$

where $\Psi(\varphi^j \wedge t_i \le t_j)$ is the strongest condition over interval endpoints implied by $\varphi^j \wedge t_i \le t_j$; this condition can be constructed using the quantifier elimination procedure for linear order. There are two rules for each of the above rules: one for comp^S and one for comp^B; the respective subqueries are compiled in a context type inherited from their parent, with the exception of the last one which always uses comp^B for the subqueries. This arrangement implements the conditional rules from Lemma 11.

New translation rules:

$$\begin{aligned}&\mathsf{comp}^S(Q\,\texttt{GROUP BY}\, x_1, \ldots, x_k) = \\ &\quad \{\texttt{SELECT}\, X_1 \,\texttt{AS}\, X_1, \ldots, X_k \,\texttt{AS}\, X_k\ Q^j\{\psi^j\}\}\end{aligned}$$

$$\mathsf{comp}^*\begin{pmatrix} Q\ \ \texttt{GROUP BY}\, a_1, \ldots, a_k, t_1, \ldots, t_l \\ \texttt{AGG}\, f^1 \,\texttt{AS}\, i^1, \ldots f^m \,\texttt{AS}\, i^m \end{pmatrix} =$$

$$\left\{\begin{array}{l} \texttt{SELECT}\, A_1, \ldots, A_k, T_1, \ldots, T_l, \\ \qquad\qquad F^1 \,\texttt{AS}\, i^1, \ldots F^m \,\texttt{AS}\, i^m \\ \texttt{FROM}\, \mathsf{N}_{a_1,\ldots,a_k}^{t_1,\ldots,t_l}[Q^i; Q^i] \\ \texttt{GROUP BY}\, A_1, \ldots, A_k, T_1, \ldots, T_l \end{array}\right\}\{\psi^i\}$$

If the aggregates f^i are either absent of consist only of the functions `MIN` and `MAX`, then we have $Q^i\{\varphi^i\}\} \in comp^S(Q)$. Otherwise, we are required to follow the duplicate-sensitive semantics in the translated sub-queries, and $Q^i\{\varphi^i\}\} \in comp^B(Q)$. Note that in the above rules, ψ^i is the projection of φ^i on the resulting attributes and F^i the translation of f^i [12].

Set operation translation rules:

$$\begin{aligned}&\mathsf{comp}^B(Q\ \texttt{UNION}\ [\texttt{ALL}]\ R) = \\ &\quad \{Q^i\{\varphi^i \wedge \neg \textstyle\bigvee_{j \in J} \psi^j\} \\ &\quad\ R^j\{\psi^j \wedge \neg \textstyle\bigvee_{i \in I} \varphi^i\} \\ &\quad\ Q'^i\ \texttt{UNION}\ [\texttt{ALL}]\, R'^j\{\varphi^i \wedge \psi^j\}\}\end{aligned}$$

$$\begin{aligned}&\mathsf{comp}^B(Q\ \texttt{EXCEPT}\ R) = \\ &\quad \{Q^i\{\varphi^i \wedge \neg \textstyle\bigvee_{j \in J} \psi^j\}, \\ &\quad\ Q'^i\ \texttt{EXCEPT}\, R'^j\{\varphi^i \wedge \psi^j\}\}\end{aligned}$$

$$\mathsf{comp}^B(Q \;\texttt{INTERSECT}\,[\texttt{ALL}]\; R) = \{Q'^i \;\texttt{INTERSECT}\,[\texttt{ALL}]\, R'^j\{\varphi^i \wedge \psi^j\}\}$$

where $Q^i\{\varphi^i\} \in \mathsf{comp}^S(Q)$, $R^j\{\psi^j\} \in \mathsf{comp}^S(R)$, $Q'^i = \mathsf{N}_X[Q^i; Q^i, R^j]$, and $R'^i = \mathsf{N}_X[R^j; Q^i, R^j]$.

The normalization can be omitted for the UNION operation. In addition, in a duplicate insensitive context, the union operation can also be translated more efficiently:

$$\mathsf{comp}^S(Q\,\texttt{UNION}\,[\texttt{ALL}]\,R) = \{(Q^i\{\varphi^i\}, R^j\{\psi^j \wedge \neg \bigvee_{i\in I} \varphi^i\}\}$$

Here, we can eliminate one of the terms of the translated union by allowing the translated expression to generate additional duplicates. However, we must ensure that the conditional queries are disjoint for other SQL/TP optimizations to work, so we include the $\{\neg \bigvee_{i\in I} \varphi^i\}$ term. An optimizer can choose whether to incorporate this term with the translation of R or Q based on estimated cost.

Intersection can also be translated more efficiently in a duplicate-insensitive context:

$$\begin{array}{l}\mathsf{comp}^S(Q\,\texttt{INTERSECT}\,[\texttt{ALL}|\texttt{IN}]\,R) = \\ \quad\{(\texttt{SELECT}\ \overline{\mathrm{sch}}(Q) \\ \quad\ \ \texttt{FROM}\ \ Q^i, R^j \\ \quad\ \ \texttt{WHERE}\ \overline{\mathrm{sch}}(Q) = \overline{\mathrm{sch}}(R))\{\varphi^i \wedge \psi^j\}\}\end{array}$$

where $Q^i\{\varphi^i\} \in \mathsf{comp}^S(Q)$, $R^j\{\psi^j\} \in \mathsf{comp}^S(R)$, and $\overline{\mathrm{sch}}(Q)$ denotes the concrete signature of Q. The expression $\overline{\mathrm{sch}}(Q) = \overline{\mathrm{sch}}(R)$ denotes the conjunction over all elements of $\overline{\mathrm{sch}}(Q)$ of equalities for data attributes and intersections[3] for temporal attributes. This translation essentially replaces an intersection with a join between Q and R on the attributes in their common signature.

Set difference (EXCEPT [IN]) also presents the opportunity for a more efficient implementation when we do not have to worry about duplicates in the right-hand side operand:

$$\begin{array}{l}\mathsf{comp}^S(Q\,\texttt{EXCEPT}\,[\texttt{IN}]\,R) = \\ \quad\{Q^i\{\varphi^i \wedge \neg \bigvee_{j\in J} \psi^j\}, \\ \quad\ (\texttt{SELECT}\ \overline{\mathrm{sch}}(Q) \\ \quad\ \ \texttt{FROM}\ \ Q'^i \\ \quad\ \ \texttt{WHERE}\ \ \texttt{NOT EXISTS}\ (\ \texttt{SELECT}\ * \\ \qquad\qquad\qquad \texttt{FROM}\ \ R^j \\ \qquad\qquad\qquad \texttt{WHERE}\ \overline{\mathrm{sch}}(Q) \subseteq \overline{\mathrm{sch}}(R))\{\varphi^i \wedge \psi^j\}\}\end{array}$$

where $Q^i\{\varphi^i\} \in \mathsf{comp}^S(Q)$, $R^j\{\psi^j\} \in \mathsf{comp}^S(R)$, and $Q'^i = \mathsf{N}_X[Q^i; Q^i, R^j]$. The expression $\overline{\mathrm{sch}}(Q) \subseteq \overline{\mathrm{sch}}(R)$ denotes the conjunction over all elements of $\overline{\mathrm{sch}}(Q)$ of equalities for data attributes and inclusions[3]for temporal attributes. It follows from the fact that since Q'^i has been normalized with respect to R^j that all interval values appearing in the answers to Q'^i must be *subsets* of those appearing in R^j. The translation then uses an anti-join to eliminate those without the need to normalize R. However, Q still has to be normalized to find all of the values in Q' that cannot match any (part of a) tuple in R'.

This translation, possibly followed by a duplicate elimination operator, can also be used in a duplicate preserving context. Similarly, in the duplicate-preserving contexts, the translations of INTERSECT IN and INTERSECT (again, followed by duplicate elimination) can be translated using SQL's WHERE EXISTS construct since the construction preserves duplicate insensitivity in the left-hand side operand. In all these cases one of the costly normalization operations has been eliminated.

Theorem 13 Let Q be an SQL/TP query. Then for every concrete database D we have

1. $Q([\![D]\!]) \overset{\text{BAG}}{=} [\![\mathsf{comp}^B(Q)(D)]\!] \triangleq \biguplus_{Q^j\{\varphi^j\}\in\mathsf{comp}^B(Q)} \sigma_{\varphi^j} [\![Q^j(D)]\!]$
2. $Q([\![D]\!]) \overset{\text{SET}}{=} [\![\mathsf{comp}^S(Q)(D)]\!] \triangleq \bigcup_{Q^j\{\varphi^j\}\in\mathsf{comp}^S(Q)} \sigma_{\varphi^j} [\![Q^j(D)]\!]$

Proof: The comp^B translation rules have been shown to be correct in previous work [12]. Each of the set-based translation rules comp^S can be seen to generate a result which contains the same distinct values as the corresponding bag-based translation rule.

We illustrate the improved translation in the following examples. The first example shows how the translation handles nested "distinct" queries.

Example 14 Decorrelation of SELECT DISTINCT and EXISTS-like queries; in SQL/TP syntax:

```
FROM     Q1 q1,
         (Q2 GROUP BY t) q2
WHERE    q1.t=q2.t
GROUP BY q1.a
```

is translated to the SQL/92 query

```
SELECT   q1.A
FROM     Q1 q1,
         (SELECT I AS I Q2) q2
WHERE    left(q1.I)<=right(q2.I)
  AND    right(q1.I)<=right(q2.I)
GROUP BY q1.A
```

where the left and right functions allow us to access the left and right endpoints of a given interval value, respectively. This query can be further "flattened" using the standard SQL rewriting rules. Without the ability to detect that the inner query operated in a duplicate insensitive context, we would have to use the $\mathsf{N}^t_{\{\}}$ normalization procedure, and generate a much more expensive query.

[3] To be written as order conditions over interval endpoints.

Similar examples can be made for other `EXISTS`-based queries which, in practice, cover a very large number of cases. In the above example, we could, however, completely omit the inner projection in the original query. The next example shows that this cannot be done in general:

Example 15 Duplicate elimination removal within a set operation:

```
(Q1 GROUP BY a AS a)
    EXCEPT (Q2 GROUP BY a AS a)
```

assuming `a` is a data attribute (but both `Q1` and `Q2` have additional temporal attributes in their signatures), can still be compiled to

```
(SELECT A AS A FROM Q1)
    EXCEPT (SELECT A AS A FROM Q2)
```

5 Conclusion

We have presented a technique that allows optimizing the translation of SQL/TP to SQL/92. The technique takes advantage of static determination of *duplicate insensitive* contexts in the original query and uses more lax translation rules that produce much more efficient SQL/92 queries as results. These more efficient queries can not be achieved by a pure SQL/TP to SQL/TP translation, nor by state-of-the-art commercial query optimizers from the SQL/92 queries resulting from the translation. The proposed technique introduces a novel approach to optimizing queries that are evaluated over an *encoded* database rather than over a standard relational representation. The proposed technique is also applicable in the more general *constraint* databases, e.g., where complex spatial objects are compactly encoded in a relational tables using a particular constraint representation.

5.1 Future work

The technique presents a new approach to query optimization. While the current application of the technique is limited to handling duplicates in SQL/TP, the approach is much more general and can be summarized as follows: Given a declarative query language that is translated to another language to facilitate query evaluation over a particular encoding of data, can the translation process take advantage of the fact that in certain query contexts we can use a translation that produces more efficient translations (but which are not equivalent to the original query in general)?

In particular, we plan to investigate how `NOT EXISTS`-like queries, the second most common class of queries in practice, can be optimized. We look for situations in which the normalization operator can be eliminated or at least simplified.

Acknowledgments. The authors gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada and the Communications and Information Technology of Ontario for their support of this research.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] M. Böhlen, R. Snodgrass, and M. Soo. Coalescing in Temporal Databases. In *International Conference on Very Large Data Bases*, pages 180–191, 1996.

[3] J. Chomicki. Temporal Query Languages: A Survey. In D. Gabbay and H. Ohlbach, editors, *Temporal Logic, First International Conference*, pages 506–534. Springer-Verlag, LNAI 827, 1994.

[4] J. Chomicki and D. Toman. Temporal Logic in Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 3, pages 31–70. Kluwer, 1998.

[5] P. Godfrey, J. Grant, J. Gryz, and J. Minker. Integrity constraints: Semantics and applications. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 9. Kluwer, 1998.

[6] V. L. Khizder, D. Toman, and G. Weddell. Reasoning about Duplicate Elimination with Description Logic. In *International Conference on Deductive and Object-Oriented Databases*, 2000.

[7] N. A. 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. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 67–91. Benjamin/Cummings, 1993.

[8] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex Query Decorrelation. In *12th International Conference on Data Engineering*, pages 450–458, 1996.

[9] R. T. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Kafer, N. Kline, K. Kulkarni, T. Y. C. Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. A. Sripada. TSQL2 language specification. *SIGMOD Record*, 23(1):65–86, Mar. 1994.

[10] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding valid time to SQL/Temporal. ISO/IEC JTC1/SC21/-WG3 DBL MAD-146r2 21/11/96, (change proposal), International Organization for Standardization, 1996.

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

[12] D. Toman. Point-based temporal extensions of SQL. In *International Conference on Deductive and Object-Oriented Databases*, 1997.

[13] D. Toman. Temporal Extensions of SQL: a Constraint Approach. In G. Kuper, L. Libkin, and J. Paradaens, editors, *Constraint Databases*, chapter 8, pages 31–70. Springer, 2000.