Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

Data & Knowledge Engineering 44 (2003) 143–164

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Optimizing temporal queries: efficient handling of duplicates

Ivan T. Bowman, David Toman [*]

Department of Computer Science, University of Waterloo, 200 University Ave. West, Waterloo, Ont., Canada N2L 3G1

Received 10 December 2001; accepted 3 July 2002

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.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Temporal query languages; Duplicate semantics for temporal queries; Compilation of temporal queries; Query optimization; Query performance

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 [10,12–15]. 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 [4].

[*] Corresponding author. Tel.: +1-519-888-4567x444; fax: +1-519-885-1208.
E-mail address: david@uwaterloo.ca (D. Toman).

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 a compact encoding of sets of time instants (often referred to as periods of validity), for example by using intervals, *bitemporal elements*, or other interval-based encoding. The chosen encoding is then used as the *concrete* domain for the representation of values of temporal attributes.

Example 1. A software development company maintains a time reporting database that records the projects and tasks that each developer worked on throughout the year. The relation used for this reporting has the following signature:

`Assignment(Day, Employee, Project, Task).`

Fig. 1 contains an instance of this relation and a compact interval encoding of the instance. Note that Ann was assigned to both tasks 1 and 2 of project 'A' during February.

Since the query semantics is defined over time instants, one way to evaluate queries would be to use a standard relational query language over the expanded, point-based instance. This approach would allow queries to be written and executed with the well-understood semantics of relational theory. However, this approach would require prohibitive time and space to answer queries since the number of tuples to be processed could depend on the *values* of attributes in the compact encoding. For example, a single interval-encoded tuple can be expanded into an arbitrary number of tuples by adjusting the interval endpoints.

Since it is not feasible to execute queries against the point-based instance, queries are ultimately evaluated over a compact encoding, independently of the query language used. The query evaluation is often based on *translating* the original query to a standard relational query that can be

Day	Employee	Project	Task
Jan 1	Ann	A	1
...	...	...	...
Feb 28	Ann	A	1
Feb 1	Ann	A	2
...	...	...	...
Mar 31	Ann	A	2
Jan 1	Bob	B	3
...	...	...	...
Jan 31	Bob	B	3

Day	Employee	Project	Task
Jan 1 – Feb 28	Ann	A	1
Feb 1 – Mar 31	Ann	A	2
Jan 1 – Jan 31	Bob	B	3

Fig. 1. Time-reporting example database.

executed over the encoding [15] or [12–14] (for snapshot and sequenced fragments). Translation allows queries to be written with a well-understood relational language, and this translation yields queries that can be executed in time and space that is at most polynomial in the size of the compact encoding, regardless of the particular values in intervals. 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.

Duplicates are not permitted in pure relational theory, but SQL permits them in order to allow counting, and also to allow applications to indicate that the query processor does not need to spend the time removing duplicates if they do not affect the application. For these reasons, SQL permits bags instead of sets, and defines the multiplicity of tuples expected from the various operations.

Example 2. Let D be a temporal database which asserts that value a is in the relation R at all time instants $t \geqslant 0$. 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) compact encodings of D. 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 [15,16] for description of the language) *prohibits* duplicate-preserving projections of temporal attributes.

Example 3. The commonly proposed solution for implementing the query from Example 2 that simply projects out the interval attribute [12] (non-sequenced 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 [5,15].

The query translator may recognize contexts within the translated query where the answer is not sensitive to the duplicates generated by a subquery. Consider the following example.

Example 4. If we are interested in the time reporting for employee Ann, we can write the query Q using the schema of Fig. 1 as follows:

```
SELECT DISTINCT Day, Project
FROM Assignment
WHERE Employee = 'Ann'
```

We can use query Q to build up larger queries. If we are interested only in knowing the projects that Ann 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 do not affect the result (for example, because of the `DISTINCT` keyword in query Q_2), then the translator could simply project out the 'Task' attribute of Q. The intermediate results of Q would then be:

Day	Project
Jan 1–Feb 28	A
Feb 1–Mar 31	A

This intermediate result contains duplicate tuples because of the overlapping time intervals; these duplicates are then eliminated by the `DISTINCT` of query Q_2. If we were also interested in the days when 'Ann' or 'Bob' was assigned to each project, with duplicates for days they were both assigned, we could write a query Q_3 as:

```
(Q) UNION ALL (SELECT DISTINCT Day, Project
               FROM Assignment
               WHERE Employee = 'Bob')
```

In this case, we must eliminate the duplicate values resulting from the overlapping time intervals. The relational projection used in the Q_2 context is not sufficient 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.

The intermediate results of Q as found by the SQL/TP normalization procedure is the following:

Day	Project
Jan 1–Jan 31	A
Feb 1–Feb 28	A
Mar 1–Mar 31	A

In this example, the context of evaluation of a query can be used to choose a more efficient evaluation technique. In the Q_2 context, Q can be evaluated efficiently using the relational duplicate-preserving project. In the Q_3 context, we must use the more expensive normalization procedure to eliminate duplicates in the range-encoded time intervals.

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 (relational) *projection* that projects out the *compact representation* of sets of time instants (as in Example 3) and thus provides a vastly better performance (similarly to duplicate preserving vs. distinct projections in SQL).

The problem of efficiently executing relational queries in the presence of duplicates has been well investigated for non-temporal query languages, and formal descriptions of algebras for bag queries have been developed which allow the issues of duplicates to be described precisely [7,9]. In addition to formal descriptions, results on optimization allow moving group-by and duplicate elimination operations to earlier or later contexts during execution plans in order to improve execution cost [3,8,17]. These optimization results also allow duplicate elimination operations to be avoided in cases where they do not affect the results. However, these prior results do not immediately generalize to the problem of executing temporal queries containing duplicates, mainly because SQL/TP supports infinite abstract databases (as shown in Example 2). In particular, solutions based on counting and aggregation to avoid problems with duplicate values in SQL/92 queries fail for SQL/TP queries as the counts would need to be infinite (cf. Example 3). In addition, to recover answers required by SQL's definition, an *unfold* [10] or *expand* [8] operation is necessary to replicate a tuple based on an *integer value* stored in the database. This operation, however, cannot be expressed in SQL itself.

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 by Toman [15,16]. However, we would like to stress that the proposed approach is applicable to other proposals, e.g., to IXRM [10] and to the SQL/Temporal–TSQL2–ATSQL2 family [12–14].

The remainder of this paper is organized as follows. Section 2 provides an introduction to the SQL/TP query language and the basic compilation technique used to translate SQL/TP queries into SQL/92 (a more detailed description can be found in [15,16]). 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 4.2 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*.

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 5 (*Abstract temporal database*). *A signature* $\mathrm{sig}(R)$ 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 $\mathrm{sig}(R_1), \ldots, \mathrm{sig}(R_n)$. 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 6 (*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 data attribute A for every data attribute a in E, and (2) an *interval* attribute I_t for every temporal attribute t in E. The attribute I_t holds the interval encoding of consecutive time instants. [1] 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 $[\![\cdot]\!]$. The meaning of a single concrete tuple $x = (I_t, a_1, \ldots, a_k)$ is a bag of tuples $[\![x]\!] = \{(t, a_1, \ldots, a_k) | t \in I_t\}$. Similarly, for concrete tuples with multiple abstract temporal attributes we define the result of the $[\![\cdot]\!]$ operator to be the hypercube resulting from allowing each temporal attribute t_i to range within its interval bounds defined in the concrete tuple. This map is extended to a concrete relation R by taking the additive union $\biguplus_{t \in R}[[t]]$ of the meanings of all concrete tuples $t \in R$.

[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.

```
<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]
```

Fig. 2. Syntax of SQL/TP.

2.1. Syntax and semantics

The syntax of the SQL/TP query language is defined by the grammar in Fig. 2, which has been slightly simplified to omit details such as the precedence of query constructs. Here, $\langle$`cexp`$\rangle$ denotes a column expression (an aggregate function), $\langle$`sexp`$\rangle$ denotes a scalar expression, $\langle$`condition`$\rangle$ represents an atomic condition, and $\langle$`setop`$\rangle$ is a multi-set operation. The meaning of $Q(D)$ is defined by induction as follows:

(1) $R(D)$ denotes the relation stored in D named R.
(2) $(\texttt{FROM}\ Q_1\ r_1, \ldots, Q_k\ r_k)(D) = Q_1(D) \times \cdots \times Q_k(D)$ 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 temporal attributes, the allowed conditions φ are of the form $a_i + c \leqslant a_j$, $c \leqslant a_i$, or $a_i \leqslant c$, where a_i and a_j are 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) $(Q\ \texttt{GROUP BY}\ a_1, \ldots, a_k\ \texttt{AGG}\ f^1(b_1)\ \texttt{AS}\ i_1, \ldots, f^n(b_n)\ \texttt{AS}\ i_n)(D)$ 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`. The aggregate functions may be omitted, in which case the operation has the effect of removing duplicates (similarly to the SQL/92 `SELECT DISTINCT`).
(6) $(Q_1 \langle\texttt{SETOP}\rangle Q_2)(D) = Q_1(D)\ op\ Q_2(D)$ are the set (bag) operators. The allowed operations *op* 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 [15,16], we introduce two additional multi-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 $\{|x \in \cdots|\}$ 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. The other set (bag) operators behave exactly like their SQL/92 counterparts.

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; cf. Example 2) 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. 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 [15].

The particular restriction to bags with *finite duplication* only, and in turn the restriction placed on the projection operation, is needed to guarantee meaningful semantics of bag operations.

Example 7. Consider the query

```
(SELECT d FROM r) EXCEPT ALL (SELECT d FROM r WHERE t > 10)
```

executed over a concrete relation $r(d,t) = \{(a,[0,\infty])\}$. Clearly, if a duplicate-preserving projection of temporal attributes (in this case t) was permitted, the two subqueries would both return countably many tuples (a), and thus the number of duplicate tuples in the result of the query (according to the usual rules for cardinal arithmetic) is not defined.

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, 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 restrictions on temporal attributes. 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:

$$Q([\![D]\!]) = [\![\mathtt{comp}(Q)(D)]\!] \stackrel{\triangle}{=} \biguplus_{Q^j\{\varphi^j\}\in\mathtt{comp}(Q)} \sigma_{\varphi^j}[\![Q^j(D)]\!]$$

The SQL/TP language carefully models the set and the multi-set semantics of SQL—SQL/TP queries 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 [15,16].

The second problem that arises during translation is that 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. The challenge here lies in the definition of relational operators that preserve semantics over the interval-based encoding.

Example 8. Let D be the following encoding of a temporal database:

$$D = \langle S_1 : \{(a,[0,4]),(b,[0,1]),(b,[7,8])\},\ S_2 : \{(a,[4,6]),(b,[1,7])\}\rangle$$

If we compute the expression $S = S_1$ EXCEPT S_2, then one possible encoding is given by

$$S : \{(a,[0,3]),(b,[0,0]),(b,[8,8])\}$$

Relational database engines do not have the ability to generate the desired results using existing relational techniques. For this purpose, we introduce a novel *normalization* technique. The idea behind the technique is quite simple: we normalize the encoded input relations in a manner which allows relational operations to treat intervals as atomic values. In this way, the ordinary relational operations give the desired results for the encoded relations. Fig. 3 shows a graphical depiction of the normalization operation applied to implement set difference for Example 8. The formal definitions are as follows:

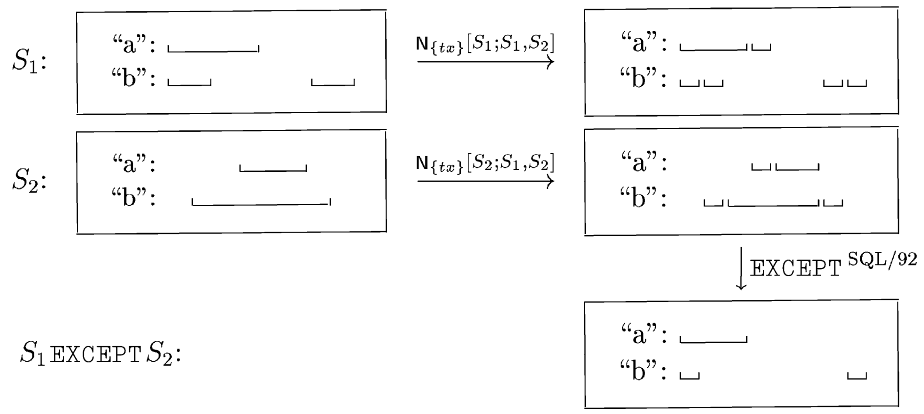

Fig. 3. Set difference using normalization.

Definition 9 (*Time compatibility*). Let $\{Q_1, \ldots, Q_k\}$ be a set of SQL/92 queries with compatible signatures and X a subset of their attributes. We call the queries $Q_1, \ldots, Q_k$ *X-compatible* if, whenever there are two concrete tuples $t_1 \in Q_i(D)$ and $t_2 \in Q_j(D)$, then

$$[\![\pi_X(t_1)]\!] \cap [\![\pi_X(t_2)]\!] \neq \emptyset \Rightarrow \pi_X(t_1) = \pi_X(t_2)$$

for all concrete temporal databases D and all $0 < i \leqslant j \leqslant k$.

The definition of an X-compatible set of queries says that if the meanings of two concrete tuples intersect then it is always the case that these two tuples coincide. This way we can guarantee the intervals behave like points with respect to set/bag operations. If X contains all of the temporal attributes of the queries, then we say that the queries are *time compatible*.

It is also easy to see that we can define a *normalization operation* that transforms an arbitrary set of queries to an X-compatible set of $[\![\cdot]\!]$-equivalent queries. Moreover, this operation can be defined using a first-order query: [2]

Lemma 10. *Let $\{Q_1, \ldots, Q_k\}$ be a set of SQL/92 queries with compatible signatures and X a subset of their attributes. Then there are first-order queries $\mathsf{N}_X[Q_i; Q_1, \ldots, Q_k]$ such that*

(1) $[\![Q_i(D)]\!] = [\![\mathsf{N}_X[Q_i; Q_1, \ldots, Q_k](D)]\!]$ *for all concrete databases D,*
(2) $\{\mathsf{N}_X[Q_i; Q_1, \ldots, Q_k] : 0 < i \leqslant k\}$ *are X-compatible.*

To define a time-compatible set of queries the above lemma is used for all temporal attributes in the common signature.

2.3. Implementation of normalization operator

The normalization operation can be performed in $O(n \log n) + O(m)$ where n is the combined size of the inputs and m is the size of the output of the operator. For example, it can be implemented by merging sorted inputs. More efficient implementations are possible in cases the optimizer knows properties of the inputs to the operator (e.g., that the inputs are already sorted). However, the normalization operation is never better than $O(n)$ in the size of the input relations, and removing it or replacing it by SQL's duplicate-preserving projection always improves performance.

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:

[2] Similarly to coalescing; a native implementation of the normalization can often be made more efficient.

Definition 11 (*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; the $=$ sign 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.

Definition 12 (*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. Here, the signature σ represents the schema of attributes expected by the context, and we say that the above queries Q are *compatible* with $C[\ ]$. For simplicity we omit the context signatures whenever possible.

For example,

$C[\ ] = ([\ ]\ \texttt{WHERE Employee} = \text{‘}\texttt{Ann}'\text{)}$ and
$C'[\ ] = (\texttt{FROM}\ Q_1\ r_1, [\ ]\ r_2)$

are contexts. The expression $C[\texttt{Assignment}]$ represents the expansion:

$(\texttt{Assignment WHERE Employee} = \text{‘}\texttt{Ann}')$

and the expression $C[(\texttt{Assignment WHERE Project} = \text{‘}\texttt{A}')]$ represents the expansion:

$((\texttt{Assignment WHERE Project} = \text{‘}\texttt{A}')\ \texttt{WHERE Employee} = \text{‘}\texttt{Ann}')$.

If $C[\ ]$ and $C'[\ ]$ are two contexts with signature σ and σ', respectively, then $C[C'[\ ]]$ is also a context of signature σ'. Here, $C[C'[\ ]]$ stands for the composition of contexts. The above contexts also define the composite context and the expansion

$C[C'[\ ]] = ((\texttt{FROM}\ Q_1 r_1, [\ ] r_2)\ \texttt{WHERE Employee} = \text{‘}\texttt{Ann}')$
$C[C'[\texttt{Assignment}]] = ((\texttt{FROM}\ Q_1 r_1, \texttt{Assignment}\ r_2)\ \texttt{WHERE Employee} = \text{‘}\texttt{Ann}')$,

respectively.

Definition 13 (*Duplicate insensitive context*). A context $C[\ ]$ is *duplicate insensitive* if

$$Q_1 \overset{\text{SET}}{=} Q_2 \Rightarrow C[Q_1] \overset{\text{BAG}}{=} C[Q_2]$$

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

Example 14. 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 15. *It is undecidable whether or not an arbitrary context $C[\ ]$ is duplicate insensitive.*

Proof. Consider the context $C[\ ] = (\texttt{FROM}[\ ]r, Qs)$ where Q is a closed query. [3] Then $C[\ ]$ is duplicate insensitive if and only if Q is unsatisfiable (empty). However, emptiness is not decidable [1], so there cannot exist a decision algorithm for duplicate insensitivity. □

Definition 16 (*Set preserving context*). A context $C[\ ]$ is *set preserving* if

$$Q_1 \overset{\text{SET}}{=} Q_2 \Rightarrow C[Q_1] \overset{\text{SET}}{=} C[Q_2]$$

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

Example 17. Clearly, every context that is duplicate insensitive is also set preserving. In addition, there are several contexts which are set preserving but which are not duplicate insensitive. Consider the context

$$C[\ ] = ([\ ]\ \texttt{WHERE}\ \varphi).$$

This context is set preserving since the result of the expression contains the same set of distinct tuples in its output for any pair of set-equivalent queries Q_1 and Q_2 with signatures compatible with $C[\ ]$.

Lemma 18 (Set preserving contexts). *The following contexts are set preserving*:

$\texttt{FROM}\ Q_1\ r_1, \ldots, [\ ]r_i, \ldots, Q_k\ r_k,$
$[\ ]\ \texttt{WHERE}\ \varphi,$
$\texttt{SELECT}\ e_1\ \texttt{AS}\ i_1, \ldots, e_k\ \texttt{AS}\ i_k\ [\ ],$
$[\ ]\ \texttt{GROUP BY}\ a_1, \ldots, a_k\ \texttt{AGG}\ f^1\ \texttt{AS}\ i^1, \ldots, f^m\ \texttt{AS}\ i^m$ (for $f^i \in \{\texttt{MIN}, \texttt{MAX}\}$),
$[\ ]\ \langle\texttt{SETOP}\rangle\ Q_2,$
$Q_1\ \langle\texttt{SETOP}\rangle\ [\ ]$

where $\langle\texttt{SETOP}\rangle$ is one of the following:

- UNION, UNION ALL,
- INTERSECT, INTERSECT IN, INTERSECT ALL,
- EXCEPT, EXCEPT IN.

Note that the EXCEPT ALL contexts are *not* set preserving in general, since the monus operation produces different *sets* as results depending on the multiplicities of the tuples in each input.

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

Proof. By case analysis.

Case `FROM`: Consider the context

$$C_F[\ ] = \texttt{FROM}\ Q\ r, [\ ]\ s$$

For any two compatible queries Q_1 and Q_2, the results of the expansion $C_F[Q_1]$ is $Q \times Q_1$ and the result of $C_F[Q_2]$ is $Q \times Q_2$. For each tuple $t \in C_F[Q_1]$, there are tuples $r \in Q$ and $s \in Q_1$ such that t is the concatenation of r and s. If $Q_1 \overset{\text{SET}}{=} Q_2$, then $s \in Q_2$ as well, so $t \in C_F[Q_2]$. Therefore, $\{s \in C_F[Q_1]\} \subseteq \{s \in C_F[Q_2]\}$. By a similar argument, $\{s \in C_F[Q_2]\} \subseteq \{s \in C_F[Q_1]\}$. Therefore,

$$Q_1 \overset{\text{SET}}{=} Q_2 \Rightarrow C_F[Q_1] \overset{\text{SET}}{=} C_F[Q_2]$$

and the context $C_F[\ ]$ is set preserving. The extension to a general Cartesian product of more than two relations follows from the standard equivalence with a sequence of binary Cartesian products and an inductive argument.

Case `WHERE`: The context

$$C_W[\ ] = [\ ]\ \texttt{WHERE}\ \varphi$$

produces the same set of distinct resulting tuples for any two set equivalent queries Q_1 and Q_2 compatible with the context. For any tuple $t \in C_W[Q_1]$, $t \in Q_1$ and hence $t \in Q_2$ and therefore $t \in C_W[Q_2]$. Thus, the context $C_W[\ ]$ is set preserving.

Case `SELECT`: The context

$$C_S[\ ] = \texttt{SELECT}\ e_1\ \texttt{AS}\ i_1, \ldots, e_k\ \texttt{AS}\ i_k\ [\ ]$$

is also set preserving. Let Q_1 and Q_2 be any two set equivalent queries compatible with $C_S[\ ]$. If tuple $t \in C_S[Q_1]$, then there is a tuple $r \in Q_1$ such that t is the projection of r. By set equivalence, $r \in Q_2$ and therefore $t \in C_S[Q_2]$. This gives the result that $C_S[\ ]$ is a set preserving context.

Case `GROUP BY`: The context

$$C_G[\ ] = [\ ]\ \texttt{GROUP BY}\ a_1, \ldots, a_k\ \texttt{AGG}\ f^1\ \texttt{AS}\ i^1, \ldots, f^m\ \texttt{AS}\ i^m$$

is duplicate insensitive if either there are no aggregate functions, or they are drawn only from `MIN` and `MAX`. In this case, the context is trivially set preserving since duplicate insensitivity is a stronger condition.

Case ⟨`setop`⟩: The contexts

$$C_{P_1}[\ ] = [\ ]\ \langle\texttt{SETOP}\rangle\ Q_2$$
$$C_{P_2}[\ ] = Q_1\ \langle\texttt{SETOP}\rangle\ [\ ]$$

are set preserving for the set operations listed above. For these contexts, the presence of a distinct tuple t in the results depends only on the existence of a *single* tuple t in one or both of the inputs (depending on the particular ⟨`setop`⟩ used). Thus, additional duplicates do not affect the distinct tuples generated in the result, although they do change the multiplicity. □

Lemma 19 (Duplicate insensitive composition). *Let $C_1[\ ]$ be a duplicate insensitive context with signature σ_1, and $C_2[\ ]$ a set-preserving context with signature σ_2. Then the context defined by composition of these two contexts, $C[\ ] = C_1[C_2[\ ]]$, is duplicate insensitive.*

Proof. For any two queries Q_1, Q_2 compatible with σ_2, we have that

$$Q_1 \overset{\text{SET}}{=} Q_2 \Rightarrow C_2[Q_1] \overset{\text{SET}}{=} C_2[Q_2]$$

because $C_2[\ ]$ is set preserving. Further, we have that

$$C_2[Q_1] \overset{\text{SET}}{=} C_2[Q_2] \Rightarrow C_1[C_2[Q_1]] \overset{\text{BAG}}{=} C_1[C_2[Q_2]]$$

since C_1 is duplicate insensitive. Thus, for any two queries Q_1, Q_2 compatible with σ_2, we have

$$Q_1 \overset{\text{SET}}{=} Q_2 \Rightarrow C[Q_1] \overset{\text{BAG}}{=} C[Q_2]$$

and $C[\ ]$ is thus duplicate-insensitive. □

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 20 (Duplicate insensitive contexts (i)). *The contexts*

$$\begin{array}{l}[\ ]\ \texttt{GROUP BY}\ a_1,\ldots,a_k\ \texttt{AGG}\ f^1\ \texttt{AS}\ i^1,\ldots,f^m\ \texttt{AS}\ i^m\ (\text{for } f^i \in \{\texttt{MIN},\texttt{MAX}\}),\\ [\ ]\ \texttt{INTERSECT}\ Q, [\ ]\ \texttt{EXCEPT}\ Q, [\ ]\ \texttt{UNION}\ Q,\\ Q\ \texttt{INTERSECT}\ [\ ], Q\ \texttt{EXCEPT}\ [\ ], Q\ \texttt{UNION}\ [\ ]\end{array}$$

are duplicate insensitive.

Proof. All the above contexts $C[\ ]$ are set preserving (by Lemma 18); further, these contexts always generate duplicate-free results. For these contexts,

$$Q_1 \overset{\text{SET}}{=} Q_2 \Rightarrow C[Q_1] \overset{\text{SET}}{=} C[Q_2]$$

and further, since duplicates are eliminated in the output,

$$C[Q_1] \overset{\text{SET}}{=} C[Q_2] \Rightarrow C[Q_1] \overset{\text{BAG}}{=} C[Q_2]$$

Thus,

$$Q_1 \overset{\text{SET}}{=} Q_2 \Rightarrow C[Q_1] \overset{\text{BAG}}{=} C[Q_2]$$

and the contexts are duplicate insensitive as desired. □

Lemma 21 (Duplicate insensitive contexts (ii)). *The contexts*

$$Q\ \texttt{INTERSECT IN}\ [\ ],\ \text{and}\ Q\ \texttt{EXCEPT IN}\ [\ ]$$

are duplicate insensitive.

Proof. Let $C[\ ] = (Q\ \langle\texttt{op}\rangle\ [\ ])$ be a context where $\langle\texttt{op}\rangle$ is `INTERSECT IN` or `EXCEPT IN`. Let Q_1 and Q_2 be any two queries compatible with the signature of context $C[\ ]$ where $Q_1 \overset{\text{SET}}{=} Q_2$. For each tuple $t \in Q$, either $t \in Q_1 \wedge t \in Q_2$ or $t \notin Q_1 \wedge t \notin Q_2$. The decision to include t in the output of $C[\ ]$ depends only on the existence of a single witness in Q_1 and Q_2, and different multiplicities does not affect this decision. Further, all duplicates of $t \in Q$ are treated identically, so the multiplicity of t in the result will either be zero or the same as the multiplicity in the original query. Therefore, we see that for any pair of queries Q_1 and Q_2

$$Q_1 \overset{\text{SET}}{=} Q_2 \Rightarrow C[Q_1] \overset{\text{BAG}}{=} C[Q_2].$$

Thus the context $C[\ ]$ is duplicate insensitive. □

Lemma 22 (Inference of duplicate insensitivity). *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
$C[[\ ]\ \texttt{UNION ALL}\ Q]$

are also duplicate insensitive.

Proof. The inner contexts in all of the above cases are set preserving by Lemma 18. Since context $C[\ ]$ is duplicate insensitive, the composed context $C[C'[\ ]]$ is also duplicate insensitive by Lemma 20. □

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. Our focus is the elimination of the *normalization* operator generated by the original approach [15] in the translations of the following two SQL/TP constructs:

- the aggregation/duplicate elimination operation and
- a set/bag operation.

More formally, let Q_1 and Q_2 be two SQL/92 queries with a common concrete signature $\overline{X}$ corresponding to an SQL/TP signature $\bar{x}$, perhaps resulting from translating parts of an SQL/TP query. The original translation establishes the following:

$$[\![Q_1(D)]\!] \texttt{ GROUP BY } \bar{y} \overset{\text{BAG}}{=} [\![\mathsf{N}_{\overline{Y}}[Q_1;Q_1] \texttt{ GROUP BY } \overline{Y}(D)]\!]$$

$$[\![Q_1(D)]\!] \ \langle\texttt{SETOP}\rangle\ [\![Q_2(D)]\!] \overset{\text{BAG}}{=} [\![\mathsf{N}_{\overline{X}}[Q_1;Q_1,Q_2] \ \langle\texttt{SETOP}\rangle\ \mathsf{N}_{\overline{X}}[Q_2;Q_1,Q_2](D)]\!]$$

for $\bar{y} \subseteq \bar{x}$ a set of grouping attributes and D an arbitrary concrete database.

It is easy to see that we cannot use optimization based on rewrites purely within SQL/TP and therefore (modified) techniques for "distinct" elimination used by standard SQL optimizers, e.g., [6,9,11], cannot be used: there is no SQL/TP expression that could replace the original one in a duplicate-insensitive context and simplify the result of the translation.

In particular the (commonly suggested) expression SELECT $\bar{x}$ AS $\bar{x}$ Q is not a valid substitute and, moreover, must be disallowed due to the problems outlined in Example 1. In addition, it is also easy to see that translating an SQL/TP expression *cannot* generate SQL's *duplicate preserving* projection since such SQL/92 queries are not $[\![\cdot]\!]$-generic over concrete temporal databases in general (while all translations of SQL/TP queries are $[\![\cdot]\!]$-generic).

Therefore, we consider *modifying* the SQL/TP-to-SQL translation instead. We proceed in two steps. First we explore translation rules in which the normalization operator can be removed if we weaken the requirement of bag equality to set equality for the results of the compiled queries. Second, we integrate these new rules with an existing SQL/TP compilation procedure [15,16] employing duplicate insensitivity inferred for query contexts present in the input query.

4.1. Relaxed compilation rules

In the following we use Q_1 and Q_2 to stand for two SQL/92 queries with a common concrete signature $\overline{X}$ corresponding to a SQL/TP signature $\bar{x}$. The following Lemmas treat the individual cases.

First and most important is the case where the GROUP BY operation in SQL/TP is used for projection. Here we want to use SQL's vastly more efficient duplicate-preserving projection. This is, however, only possible in duplicate insensitive contexts:

Lemma 23 (Aggregation and duplicate elimination). *Let D be an arbitrary concrete database and $\bar{y}$ a subset of Q_1's attributes. Then*

$$[\![Q_1(D)]\!] \texttt{ GROUP BY } \bar{y} \overset{\text{SET}}{=} [\![\texttt{SELECT } \overline{Y} \texttt{ FROM } Q_1(D)]\!]$$

Proof. Let $\bar{a}$ be an abstract tuple such that $\bar{a} \in [\![Q_1(D)]\!]$ GROUP BY $\bar{y}$. Then there must be a tuple $\overline{ab} \in [\![Q_1(D)]\!]$ and consequently a concrete tuple $\overline{AB} \in Q_1(D)$ such that $\overline{ab} \in [\![\overline{AB}]\!]$. Then, however, $\bar{A} \in$ SELECT $\overline{Y}$ FROM $Q_1(D)$ and thus $\bar{a} \in [\![\texttt{SELECT } \overline{Y} \texttt{ FROM } Q_1(D)]\!]$ since $\bar{a} \in [\![\bar{A}]\!]$. The proof of the reverse inclusion is similar. □

The right-hand side query is a pure SQL/92 query and does not involve the normalization operator. Thus, we use this equation to generate a translation rule (cf. Section 4.2). Note also that the query on the right-hand side is not bag-equivalent to *any* SQL/TP query. The following three Lemmas provide similar results for the other operations that involve normalization:

Lemma 24 (Union). *Let D be an arbitrary concrete database. Then*

$$[\![Q_1(D)]\!]\ \texttt{UNION [ALL]}\ [\![Q_2(D)]\!] \overset{\text{SET}}{=} [\![Q_1\ \texttt{UNION ALL}\ Q_2(D)]\!]$$

Proof. Let $\bar{a} \in [\![Q_1(D)]\!]$ UNION [ALL] $[\![Q_2(D)]\!]$. Then $\bar{a}$ must belong to the answer to Q_1 or Q_2. W.l.o.g., we assume $\bar{a} \in [\![Q_1(D)]\!]$. Then, there must be a concrete tuple $\bar{A} \in Q_1(D)$ such that $\bar{a} \in [\![\bar{A}]\!]$. Since $\bar{A} \in Q_1$ UNION ALL $Q_2(D)$ we have $\bar{a} \in [\![Q_1$ UNION ALL $Q_2(D)]\!]$. Again, the other direction is similar. □

In duplicate insensitive contexts the set/bag intersection can be turned into a join:

Lemma 25 (Intersection). *Let D be an arbitrary concrete database. Then*

$$[\![Q_1(D)]\!]\ \texttt{INTERSECT [ALL|IN]}\ [\![Q_2(D)]\!] \overset{\text{SET}}{=} [\![\texttt{SELECT}\ r.\overline{X} \cap s.\overline{X}\ \texttt{FROM}\ Q_1 r, Q_2 s\ \texttt{WHERE}\ r.\overline{X} \cap s.\overline{X} \neq \emptyset(D)]\!]$$

where $r.\overline{X} \cap s.\overline{X}$ is a select list consisting of $r.X_j$ AS X_j for data attributes and $r.I_j \cap s.I_j$ AS I_j for temporal attributes. The select list expression $r.I_j \cap s.I_j$ AS I_j gives the interval encoding of the overlap of the $r.I_j$ and $s.I_j$ intervals. Similarly, $r.\overline{X} \cap s.\overline{X} \neq \emptyset$ is a conjunction of equalities for matching data attributes and conditions of the form $r.I_j \cap s.I_j \neq \emptyset$ for temporal attributes. This latter condition identifies tuples r and s that have overlapping interval encodings of the I_j temporal attribute. This condition can, of course, be further expressed as a simple order condition involving interval endpoints.

Proof. If $\bar{a} \in [\![Q_1(D)]\!]$ INTERSECT [ALL|IN]$[\![Q_2(D)]\!]$ then there must be concrete tuples $A_1 \in Q_1(D)$ and $A_2 \in Q_2(D)$ such that $\bar{a} \in [\![\bar{A}_1]\!] \cap [\![\bar{A}_2]\!]$. These two tuples, however, satisfy the WHERE clause on the right-hand side of the equation and therefore a concrete tuple $\bar{A}$ representing the intersection of $\bar{A}_1$ and $\bar{A}_2$ is in the answer to the right-hand side query. Subsequently, $\bar{a}$ is in the abstract answer to the right-hand side, since $\bar{a} \in [\![\bar{A}]\!] = [\![\bar{A}_1]\!] \cap [\![\bar{A}_2]\!]$. Similarly, the other inclusion holds. □

Similarly, the set/not-exists-like set difference can be turned into an anti-join:

Lemma 26 (Difference). *Let D be an arbitrary concrete database. Then*

$$[\![Q_1(D)]\!]\ \texttt{EXCEPT [IN]}\ [\![Q_2(D)]\!] \overset{\text{BAG}}{=} [\![\texttt{SELECT [DISTINCT]}\ r.\overline{X}\ \texttt{FROM}\ \mathsf{N}_{\overline{X}}[Q_1; Q_1, Q_2]r\ \texttt{WHERE NOT EXISTS (SELECT*}\ \texttt{FROM}\ Q_2 s\ \texttt{WHERE}\ r.\overline{X} \subseteq s.\overline{X})\ (D)]\!]$$

where $r.\overline{X} \subseteq s.\overline{X}$ is a conjunction of equalities for data attributes (with matching names) and interval inclusion conditions for temporal attributes, respectively.

Proof. Let $\bar{a} \in [\![Q_1(D)]\!]$ EXCEPT[IN] $[\![Q_2(D)]\!]$. Then there must be a concrete tuple $\bar{A} \in \mathsf{N}_{\overline{X}}[Q_1; Q_1, Q_2]$ such that $\bar{a} \in [\![\bar{A}]\!]$ such that $\bar{a} \in [\![\bar{A}]\!]$ (such a tuple must exist since the normalization operation preserves meaning of abstract relations and $\bar{a} \in [\![Q_1(D)]\!]$). Moreover, there is no tuple $\overline{B} \in Q_2(D)$ such that $\bar{a} \in [\![B]\!]$. Then, however, the $\bar{A}$ tuple satisfies the NOT EXISTS condition as, due to the normalization operation applied on $Q_1(D)$ we have $[\![\bar{A}]\!] \cap [\![\overline{B}]\!] \neq \emptyset$ implies $[\![\bar{A}]\!] \subseteq [\![\overline{B}]\!]$. The other direction is the same. □

Note that in the case of set difference we can eliminate only the normalization operation connected with the R operand. On the other hand, due to the (necessary) N operator applied on Q_1, this translation preserves bag equality. [4]

4.2. 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) developed in last section into the SQL/TP-to-SQL/92 translator. We use two mutually recursive functions $\mathtt{comp}^S$ and $\mathtt{comp}^B$ that are used in the duplicate insensitive (set) and duplicate sensitive (bag) contexts, respectively ($\mathtt{comp}^*$ stands for both cases). We follow a convention that if Q^j and φ^j appear in the translation, then they range over all j such that $Q^j\{\varphi^j\} \in \mathtt{comp}^X(Q)$ where X (the appropriate bag- or set-based translation rule) is determined by the context created by the operation for its arguments (cf. Lemmas 20–22). Fig. 4 summarizes the translation rules.

Theorem 27. *Let Q be an SQL/TP query. Then for every concrete database D we have*

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

Proof. The $\mathtt{comp}^B$ translation rules have been shown to be correct in previous work [15]. Each of the set-based translation rules $\mathtt{comp}^S$ is either an instance of an equality shown in Lemmas 23–26 or follows from Lemmas 20–22. □

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

Example 28. Decorrelation of SELECT DISTINCT and EXISTS-like queries; in SQL/TP syntax:

```
FROM    Q1q1,
        (Q2 GROUP BY t) q2
WHERE   q1.t = q2.t
GROUP BY q1.a
```

[4] The set EXCEPT operator is paired with the SELECT DISTINCT right-hand side.

$$
\begin{array}{lll}
\mathtt{comp}^{*}(R) & = \{ & \mathtt{SELECT} * \mathtt{FROM}\ R\{\mathit{true}\}\ \} \\
\mathtt{comp}^{*}(\mathtt{FROM}\ Q_1 R_1, \ldots, Q_k R_k) & = \{ & \mathtt{SELECT} * \\
 & & \mathtt{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}\}\ \} \\
\mathtt{comp}^{*}(Q\ \mathtt{WHERE}\ \psi) & = \{ & \mathtt{SELECT} * \mathtt{FROM}\ Q^j \\
 & & \mathtt{WHERE}\ \Psi(\varphi^j \wedge \psi)\{\varphi^j \wedge \psi\}\ \} \\
\mathtt{comp}^{*}(\mathtt{SELECT}\ e_j\ \mathtt{AS}\ i_j, \ldots, e_k\ \mathtt{AS}\ i_k\ Q) & = \{ & \mathtt{SELECT}\ E_1\ \mathtt{AS}\ i_1, \ldots, E_k\ \mathtt{AS}\ i_k \\
 & & \mathtt{FROM}\ Q^j\{\psi^j\}\ \} \\
\mathtt{comp}^{S}(Q\ \mathtt{GROUP\ BY}\ x_1, \ldots, x_k) & = \{ & \mathtt{SELECT}\ X_1\ \mathtt{AS}\ X_1, \ldots, X_k\ \mathtt{AS}\ X_k \\
 & & \mathtt{FROM}\ Q^j\{\psi^j\}\ \} \\
\mathtt{comp}^{*}(Q\ \mathtt{GROUP\ BY}\ \bar{y}\ \mathtt{AGG}\ f^j\ \mathtt{AS}\ x^j) & = \{ & \mathtt{SELECT}\ \bar{Y}, F^j\ \mathtt{AS}\ X^j \\
 & & \mathtt{FROM}\ \mathsf{N}_{\bar{y}}[Q^i; Q^i]\ \mathtt{GROUP\ BY}\ \bar{Y}\ \{\psi^i\}\ \} \\
\mathtt{comp}^{*}(Q\ \mathtt{UNION\ ALL}\ R) & = \{ & Q^i\{\varphi^i \wedge \neg \bigvee_{j \in J} \psi^j\}, \\
 & & R^j\{\psi^j \wedge \neg \bigvee_{i \in I} \varphi^i\}, \\
 & & Q^i\ \mathtt{UNION\ ALL}\ R^j\{\varphi^i \wedge \psi^j\}\ \} \\
\mathtt{comp}^{S}(Q\ \mathtt{UNION}\ R) & = \{ & Q^i\{\varphi^i \wedge \neg \bigvee_{j \in J} \psi^j\}, \\
 & & R^j\{\psi^j \wedge \neg \bigvee_{i \in I} \varphi^i\}, \\
 & & Q^i\ \mathtt{UNION\ ALL}\ R^j\{\varphi^i \wedge \psi^j\}\ \} \\
\mathtt{comp}^{B}(Q\ \mathtt{UNION}\ R) & = \{ & Q^i\{\varphi^i \wedge \neg \bigvee_{j \in J} \psi^j\} \\
 & & R^j\{\psi^j \wedge \neg \bigvee_{i \in I} \varphi^i\} \\
 & & \mathsf{N}_X[Q^i; Q^i, R^j] \\
 & & \quad \mathtt{UNION\ ALL}\ \mathsf{N}_X[R^j; Q^i, R^j]\{\varphi^i \wedge \psi^j\} \\
\mathtt{comp}^{*}(Q\ \mathtt{EXCEPT}\ R) & = \{ & Q^i\{\varphi^i \wedge \neg \bigvee_{j \in J} \psi^j\}, \\
 & & \mathtt{SELECT\ DISTINCT}\ s.\bar{X} \\
 & & \mathtt{FROM}\ \mathsf{N}_{\bar{X}}[Q^i; Q^i, R^j]\, s \\
 & & \mathtt{WHERE\ NOT\ EXISTS(} \\
 & & \qquad \mathtt{SELECT} * \mathtt{FROM}\ R^j\ r \\
 & & \qquad \mathtt{WHERE}\ r.\bar{X} \subseteq s.\bar{X}) \\
\mathtt{comp}^{*}(Q\ \mathtt{EXCEPT\ IN}\ R) & = \{ & Q^i\{\varphi^i \wedge \neg \bigvee_{j \in J} \psi^j\}, \\
 & & \mathtt{SELECT}\ s.\bar{X}\ \mathtt{FROM}\ \mathsf{N}_{\bar{X}}[Q^i; Q^i, R^j]\, s \\
 & & \mathtt{WHERE\ NOT\ EXISTS(} \\
 & & \qquad \mathtt{SELECT} * \mathtt{FROM}\ R^j\ r \\
 & & \qquad \mathtt{WHERE}\ r.\bar{X} \subseteq s.\bar{X})\{\varphi^i \wedge \psi^j\}\} \\
\mathtt{comp}^{*}(Q\ \mathtt{EXCEPT\ ALL}\ R) & = \{ & Q^i\{\varphi^i \wedge \neg \bigvee_{j \in J} \psi^j\}, \\
 & & \mathsf{N}_X[Q^i; Q^i, R^j] \\
 & & \quad \mathtt{EXCEPT\ ALL}\ \mathsf{N}_X[R^j; Q^i, R^j]\{\varphi^i \wedge \psi^i\} \\
\mathtt{comp}^{S}(Q\ \mathtt{INTERSECT[ALL]}\ R) & = \{ & \mathtt{SELECT}\ \overline{\mathtt{sch}}(Q) \cap \overline{\mathtt{sch}}(R) \\
 & & \mathtt{FROM}\ \ Q^i, R^j \\
 & & \mathtt{WHERE}\ \overline{\mathtt{sch}}(Q) \cap \overline{\mathtt{sch}}(R) \neq \emptyset\{\varphi^i \wedge \psi^j\}\} \\
\mathtt{comp}^{B}(Q\ \mathtt{INTERSECT[ALL]}\ R) & = & \mathsf{N}_X[Q^i; Q^i, R^j] \\
 & & \quad \mathtt{INTERSECT\ ALL}\ \mathsf{N}_X[R^j; Q^i, R^j]\{\varphi^i \wedge \psi^i\}
\end{array}
$$

Fig. 4. Complete SQL/TP translation rules.

is translated to the SQL/92 query

```
SELECT    q1.a
FROM      Q1q1,
          (SELECT I_t AS I_t Q2) q2
WHERE     left(q1.I_t) <= right(q2.I_t)
AND       left(q2.I_t) <= right(q1.I_t)
GROUP BY  q1.a
```

where the `left` and `right` (perhaps user-defined) 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.

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 29. 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 for 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 cannot 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)?

Acknowledgements

The authors gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada, the Communications and Information Technology of Ontario, and iAnywhere Solutions (a Sybase company) for their support of this research.

References

[1] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.
[2] M. Böhlen, R.T. Snodgrass, M.D. Soo, Coalescing in temporal databases, in: International Conference on Very Large Data Bases, 1996, pp. 180–191.
[3] S. Chaudhuri, K. Shim, Including group-by in query optimization, in: International Conference on Very Large Data Bases, 1994, pp. 354–366.
[4] J. Chomicki, Temporal query languages: A survey, in: D.M. Gabbay, H.J. Ohlbach (Eds.), Temporal Logic, First International Conference, Springer-Verlag, 1994, pp. 506–534, LNAI 827.
[5] J. Chomicki, D. Toman, Temporal logic in information systems, in: J. Chomicki, G. Saake (Eds.), Logics for Databases and Information Systems, Kluwer, 1998, pp. 31–70 (Chapter 3).
[6] P. Godfrey, J. Grant, J. Gryz, J. Minker, Integrity constraints: Semantics and applications, in: J. Chomicki, G. Saake (Eds.), Logics for Databases and Information Systems, Kluwer, 1998, pp. 265–306 (Chapter 9).
[7] S. Grumbach, T. Milo, Towards tractable algebras for bags, Journal of Computer and System Sciences 52 (3) (1996) 570–588.
[8] A. Gupta, V. Harinarayan, D. Quass, Aggregate-query processing in data warehousing environments, in: International Conference on Very Large Data Bases, 1995, pp. 358–369.
[9] V.L. Khizder, D. Toman, G. Weddell, Reasoning about duplicate elimination with description logic, in: Computational Logic 2000, 2000, pp. 1017–1032.
[10] 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, R. Snodgrass (Eds.), Temporal Databases: Theory, Design, and Implementations, Benjamin/Cummings, 1993, pp. 67–91.
[11] P. Seshadri, H. Pirahesh, T.Y. Cliff Leung, Complex query decorrelation, in: 12th International Conference on Data Engineering, 1996, pp. 450–458.
[12] R.T. Snodgrass, M.H. Böhlen, C.S. Jensen, 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.
[13] R.T. Snodgrass, C.S. Jensen, M.H. Böhlen, Evaluating and Enhancing the Completeness of TSQL2, Technical Report TR 95-5, Computer Science Department, University of Arizona, 1995.
[14] 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, S.A. Sripada, TSQL2 language specification, SIGMOD Record 23 (1) (1994) 65–86.
[15] D. Toman, Point-based temporal extensions of SQL, in: International Conference on Deductive and Object-Oriented Databases, 1997, pp. 103–121.
[16] D. Toman, Temporal extensions of SQL: A constraint approach, in: G. Kuper, L. Libkin, J. Paradaens (Eds.), Constraint Databases, Springer, 2000, pp. 31–70 (Chapter 8).
[17] W.P. Yan, P. Larson, Eager aggregation and lazy aggregation, in: International Conference on Very Large Data Bases, 1995, pp. 345–357.

Ivan T. Bowman is a Ph.D. student in the School of Computer Science at the University of Waterloo, and a member of the Adaptive Server Anywhere query processing team at iAnywhere Solutions, a partially owned subsidiary of Sybase Inc. He is also a member of the ACM SIGMOD, and his research interests include query optimization and execution.

David Toman received his B.S. and M.S. degrees from the Masaryk University, Brno, Czech Republic in 1992, and his Ph.D. fron Kansas State University in 1996, all in computer science. He is currently assistant professor at the School of Computer Science, University of Waterloo, Canada. His research interests include temporal, deductive, and constraint database systems, query optimization and compilation, query processing in embedded control programs, description logics, logic programming, and programming languages.