

# Qex: Symbolic SQL Query Explorer

Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux  
Microsoft Research, Redmond, WA, USA  
{margus,nikolait,jhalleux}@microsoft.com

## Abstract

We describe a technique and a tool called Qex for generating input tables and parameter values for a given parameterized SQL query. The evaluation semantics of an SQL query is translated into a specific background theory for a satisfiability modulo theories (SMT) solver as a set of equational axioms. Symbolic evaluation of a goal formula together with the background theory yields a model from which concrete tables and values are extracted. We use the SMT solver Z3 in the concrete implementation of Qex and provide an evaluation of its performance.

## 1 Introduction

The original motivation behind Qex comes from *unit testing* of relational databases, where a key challenge is the automatic generation of input tables and parameters for a given query and a given test condition, where a typical test condition is that the result of the query is a nonempty table. An early prototype of Qex as a proof-of-concept and an integration of Qex into the Visual Studio Database edition is discussed in [23, 28]. Here we present a new approach for encoding queries that uses algebraic data types and equational axioms, taking advantage of recent advances in SMT technology. The encoding is much simpler than the one described in [28], and boosted the performance of Qex by several orders of magnitude. In [28] algebraic data types were not available and queries were encoded into an intermediate background theory  $\mathcal{F}^\Sigma$  using bags and a summation operator. The resulting formula was eagerly expanded, for a given size of the database, into a quantifier free formula that was then asserted to the SMT solver. The expansion often caused an exponential blowup in the size of the expanded formula, even when some parts of the expansion were irrelevant with respect to the test condition. The new approach not only avoids the eager expansion but avoids also the need for nonlinear constraints that arise when dealing with multiplicities of rows in bags and aggregates over bags. Moreover, the axiomatic approach makes it possible to encode frequently occurring like-patterns through an automata based technique, and other string constraints. To this end, Qex now encodes strings faithfully as character sequences, whereas in [28] strings were abstracted to integers with no support for general string operations. Furthermore, algebraic data types provide a straightforward encoding for value types that allow null. In addition, Qex now also handles table constraints and uses symmetry breaking formulas for search space reduction.

The core idea is as follows. A given SQL query  $q$  is translated into a term  $\llbracket q \rrbracket$  over a rich background theory that comes with a collection of built-in (predefined) functions. Tables are represented by lists of tuples, where lists are built-in algebraic data types. In addition to built-in functions (such as arithmetical operations) the term  $\llbracket q \rrbracket$  may also use functions whose meaning is governed by a set of additional axioms referred to as  $Th(q)$ . These custom axioms describe the evaluation rules of SQL queries and are in most cases defined as recursive list axioms that resemble functional programs. Table 1 provides a rough overview of the SQL constructs supported in Qex and the corresponding theories used for mapping a given construct into a formula for Z3 [30, 10] that is used as the underlying SMT solver in the implementation of Qex. As indicated in the table, in all of the cases there is also an additional set of custom axioms that are used in addition to the built-in ones.

For input tables and other parameters, the term  $\llbracket q \rrbracket$  uses uninterpreted constants. Given a condition  $\varphi$  over the result of  $\llbracket q \rrbracket$ , e.g.,  $\llbracket q \rrbracket \neq nil$  ( $\llbracket q \rrbracket$  is nonempty),  $\varphi$  is asserted to the SMT solver as a goal formula and  $Th(q)$  is asserted to the SMT solver as an additional set of axioms, sometimes called a *soft*

Table 1: Overview of features in Qex and related use of SMT theories.

Features	Built-in theories						Custom theories
	Arithmetic	Bitvectors	Sets	Arrays	Algebraic d.t.	Tuples	
Table constraints	✓		✓		✓	✓	✓
SELECT clauses	✓				✓		✓
Aggregates	✓		✓	✓	✓	✓	✓
LIKE patterns		✓			✓		✓
Null					✓		✓

*theory*. Next, a satisfiability check is performed together with *model generation*. If  $\varphi$  is satisfiable then the generated model is used to extract concrete values (interpretations) for the input table constants and other additional parameter constants.

The rest of the paper is structured as follows. Section 2 introduces some basic notions that are used throughout the paper. Section 3 defines a custom theory of axioms over lists that are used in Section 4 to translate queries into formulas. Section 5 discusses the implementation of Qex with a focus on its interaction with Z3. Section 6 provides some experimental evaluation of Qex. Section 7 is about related work, and Section 8 provides some final remarks.

## 2 Preliminaries

We assume that the reader is familiar with elementary concepts in logic and model theory, our terminology is consistent with [15] in this regard.

We are working in a fixed multi-sorted universe  $\mathcal{U}$  of values. For each sort  $\sigma$ ,  $\mathcal{U}^\sigma$  is a separate subuniverse of  $\mathcal{U}$ . The basic sorts needed in this paper are the Boolean sort  $\mathbb{B}$ , ( $\mathcal{U}^\mathbb{B} = \{true, false\}$ ), the integer sort  $\mathbb{Z}$ , and the  $n$ -tuple sort  $\mathbb{T}\langle\sigma_0, \dots, \sigma_{n-1}\rangle$  for  $n \geq 1$  of some given basic sorts  $\sigma_i$  for  $i < n$ . We also use other sorts but they are introduced at the point when they are first needed.

There is a collection of functions with a fixed meaning associated with the universe, e.g., arithmetical operations over  $\mathcal{U}^\mathbb{Z}$ . These functions and the corresponding function symbols are called *built-in*. Each function symbol  $f$  of arity  $n \geq 0$  has a fixed domain sort  $\sigma_0 \times \dots \times \sigma_{n-1}$  and a fixed range sort  $\sigma$ ,  $f : \sigma_0 \times \dots \times \sigma_{n-1} \rightarrow \sigma$ . For example, there is a built-in *relation* or *predicate* (Boolean function) symbol  $< : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{B}$  that denotes the standard order on integers. One can also declare *fresh* (new) *uninterpreted* function symbols  $f$  of arity  $n \geq 0$ , for a given domain sort and a given range sort. Using model theoretic terminology, these new symbols *expand* the signature.

*Terms* and *formulas* (or Boolean terms) are defined by induction as usual and are assumed to be well-sorted. We write  $FV(t)$  for the set of free variables in a term (or formula)  $t$ . A term or formula without free variables is *closed*.

A *model* is a mapping from function symbols to their interpretations (values). The built-in function symbols have the same interpretation in all models that we are considering, keeping that in mind, we may omit mentioning them in a model. A model  $M$  *satisfies* a closed formula  $\varphi$ ,  $M \models \varphi$ , if it provides an interpretation for all the uninterpreted function symbols in  $\varphi$  that makes  $\varphi$  true. For example, let  $f : \mathbb{Z} \rightarrow \mathbb{Z}$  be an uninterpreted function symbol and  $c : \mathbb{Z}$  be an uninterpreted constant. Let  $M$  be a model where  $c^M$  (the interpretation of  $c$  in  $M$ ) is 0 and  $f^M$  is a function that maps all values to 1. Then  $M \models 0 < f(c)$  but  $M \not\models 0 < c$ .

A closed formula  $\varphi$  is *satisfiable* if it has a model. A formula  $\varphi$  with  $FV(\varphi) = \bar{x}$  is *satisfiable* if its

existential closure  $\exists \bar{x}\varphi$  is satisfiable. We write  $\models_{\mathcal{M}} \varphi$ , or  $\models \varphi$ , if  $\varphi$  is *valid* (true in all models). Some examples:  $0 < 1 \wedge 2 < 10$  is valid;  $4 < x \wedge x < 5$ , where  $x:\mathbb{Z}$  is a free variable, is unsatisfiable because there exists no integer between 4 and 5;  $0 < x \wedge x < 3$ , where  $x:\mathbb{Z}$  is a free variable, is satisfiable.

### 3 Equational axioms over lists

The representation of a table in Qex is a *list* of rows, where a row is a tuple. While bags of rows rather than lists would model the semantics of SQL more directly (order of rows is irrelevant, but multiple occurrences of the same row are relevant), the inductive structure of a list provides a way to define the evaluation semantics of queries by recursion. The mapping of queries to axioms, discussed in Section 4, uses a collection of axioms over lists that are defined next. Intuitively, the axioms correspond to definitions of standard (higher order) functionals that are typical in functional programming. The definitions of the axioms below, although more concise, correspond precisely to their actual implementation in Qex using the Z3 API. Before describing the actual axioms, we explain the intuition behind a particular kind of axioms, that we call *equational*, when used in an SMT solver.

#### 3.1 Equational axioms and $E$ -matching in SMT solvers

During proof search in an SMT solver, axioms are triggered by matching subexpressions in the goal. Qex uses particular kinds of axioms, all of which are equations of the form

$$\forall \bar{x}(t_{\text{lhs}} = t_{\text{rhs}}) \quad (1)$$

where  $FV(t_{\text{lhs}}) = \bar{x}$  and  $FV(t_{\text{rhs}}) \subseteq \bar{x}$ . The left-hand-side  $t_{\text{lhs}}$  of (1) is called the *pattern* of (1).

*While SMT solvers support various kinds of patterns in general, in this paper we use the convention that the pattern of an equational axiom is always its left-hand-side.*

The high-level idea behind  $E$ -matching is as follows. The axiom (1) is *triggered* by the current goal  $\psi$  of the solver, if  $\psi$  contains a subterm  $u$  and there exists a substitution  $\theta$  such that  $u =_E t_{\text{lhs}}\theta$ , i.e.,  $u$  *matches the pattern* of the axiom (modulo the built-in theories  $E$ ). If (1) is triggered, then the current goal is replaced by the *logically equivalent* formula where  $u$  has been replaced by  $t_{\text{rhs}}\theta$ .

Thus, the axioms that are used in Qex can be viewed as “rewrite rules”, and each application of an axiom preserves the logical equivalence to the original goal. As long as there exists an axiom in the current goal that can be triggered, then triggering is guaranteed. Thus, termination is in general not guaranteed in the presence of (mutually) recursive axioms. Note that, unlike in term rewrite systems, there is no notion of term orderings or well-defined customizable strategies (at least not in the current version of Z3) that could be used to guide the triggering process of the axioms.

#### 3.2 Axioms over lists

For each sort  $\sigma$  there is a built-in *list sort*  $\mathbb{L}\langle\sigma\rangle$  and a corresponding subuniverse  $\mathcal{U}^{\mathbb{L}\langle\sigma\rangle}$ . (In Z3, lists are provided as built-in algebraic data types and are associated with standard constructors and accessors.) For a given element sort  $\sigma$  there is an empty list *nil* (of sort  $\mathbb{L}\langle\sigma\rangle$ ) and if  $e$  is an element of sort  $\sigma$  and  $l$  is a list of sort  $\mathbb{L}\langle\sigma\rangle$  then *cons*( $e, l$ ) is a list of sort  $\mathbb{L}\langle\sigma\rangle$ . The accessors are, as usual, *hd* (head) and *tl* (tail). In the following consider a fixed element sort  $\sigma$ . Observe that one can define a well-ordering such that, in all of the recursive cases of the axioms, the right-hand-side decreases with respect to that ordering, which guarantees that triggering terminates and implies that the axioms are well-defined. In all of the cases, the use of the list constructors in the patterns is fundamental. In most cases one can provide more

compact and logically equivalent definitions of the axioms where the right-hand-sides are combined in a disjunction, but where the pattern is too general and may cause nontermination of axiom triggering in an SMT solver.

### 3.2.1 Filter

Let  $\varphi$  be a formula with a single free variable  $x_0 : \sigma$ . Declare the function symbol  $Filter[\varphi] : \mathbb{L}\langle\sigma\rangle \rightarrow \mathbb{L}\langle\sigma\rangle$  and define the following axioms:

$$\begin{aligned} Filter[\varphi](nil) &= nil \\ \forall x_0 x_1 (Filter[\varphi](cons(x_0, x_1)) &= Ite(\varphi, cons(x_0, Filter[\varphi](x_1)), Filter[\varphi](x_1))) \end{aligned}$$

The *Ite*-term  $Ite(\phi, t_1, t_2)$  equals  $t_1$ , if  $\phi$  is true; it equals  $t_2$ , otherwise. *Ite* is a built-in function.

### 3.2.2 Map

Let  $t : \rho$  be a term with a single free variable  $x_0 : \sigma$ . Declare the function symbol  $Map[t] : \mathbb{L}\langle\sigma\rangle \rightarrow \mathbb{L}\langle\rho\rangle$  and define:

$$\begin{aligned} Map[t](nil) &= nil \\ \forall x_0 x_1 (Map[t](cons(x_0, x_1)) &= cons(t, Map[t](x_1))) \end{aligned}$$

### 3.2.3 Reduce

Let  $t : \rho$  be a term with two free variables  $x_0 : \sigma$  and  $x_1 : \rho$ . Declare the function symbol  $Reduce[t] : \mathbb{L}\langle\sigma\rangle \times \rho \rightarrow \rho$  and define:

$$\begin{aligned} \forall x (Reduce[t](nil, x) &= x) \\ \forall x_0 x_1 x_2 (Reduce[t](cons(x_0, x_2), x_1) &= Reduce[t](x_2, t)) \end{aligned}$$

For example, if  $l : \mathbb{L}\langle\mathbb{Z}\rangle$  is a list of integers, then  $Reduce[x_0 + x_1](l, 0)$  is equal to the sum of the integers in  $l$ , or 0 if  $l$  is empty (in any model that satisfies the corresponding  $Reduce[]$ -axioms).

### 3.2.4 Cross product

Declare the function symbols  $Cross : \mathbb{L}\langle\sigma\rangle \times \mathbb{L}\langle\rho\rangle \rightarrow \mathbb{L}\langle\mathbb{T}\langle\sigma, \rho\rangle\rangle$  and  $Cr : \sigma \times \mathbb{L}\langle\sigma\rangle \times \mathbb{L}\langle\rho\rangle \times \mathbb{L}\langle\rho\rangle \rightarrow \mathbb{L}\langle\mathbb{T}\langle\sigma, \rho\rangle\rangle$ , and define

$$\begin{aligned} \forall x (Cross(nil, x) &= nil) \\ \forall x (Cross(x, nil) &= nil) \\ \forall \bar{x} (Cross(cons(x_0, x_1), cons(x_2, x_3)) &= Cr(x_0, x_1, cons(x_2, x_3), cons(x_2, x_3))) \\ \forall \bar{x} (Cr(x_0, x_1, nil, x_2) &= Cross(x_1, x_2)) \\ \forall \bar{x} (Cr(x_0, x_1, cons(x_2, x_3), x_4) &= cons(T(x_0, x_2), Cr(x_0, x_1, x_3, x_4))) \end{aligned}$$

where  $T : \sigma \times \rho \rightarrow \mathbb{T}\langle\sigma, \rho\rangle$  is the built-in tuple constructor (for the given sorts). For example, the term  $Cross(cons(1, cons(2, nil)), cons(3, cons(4, nil)))$  is equal to the term

$$cons(T(1, 3), cons(T(1, 4), cons(T(2, 3), cons(T(2, 4), nil))))).$$

### 3.2.5 Remove duplicates

The function *RemoveDuplicates* is used to remove duplicates from a list. The definition makes use of built-in sets and set operations; the set sort of element sort  $\sigma$  is denoted  $\mathbb{S}\langle\sigma\rangle$ .

Declare: *RemoveDuplicates*:  $\mathbb{L}\langle\sigma\rangle \rightarrow \mathbb{L}\langle\sigma\rangle$ , *Rd*:  $\mathbb{L}\langle\sigma\rangle \times \mathbb{S}\langle\sigma\rangle \rightarrow \mathbb{L}\langle\sigma\rangle$ . Define:

$$\begin{aligned} \forall x (RemoveDuplicates(x) &= Rd(x, \emptyset)) \\ \forall x (Rd(nil, x) &= nil) \\ \forall \bar{x} (Rd(cons(x_0, x_1), x_2) &= Ite(x_0 \in x_2, Rd(x_1, x_2), cons(x_0, Rd(x_1, \{x_0\} \cup x_2)))) \end{aligned}$$

### 3.2.6 Select with grouping and aggregates

Select clauses with aggregates and grouping are translated into formulas using the following axioms. Each aggregate function  $\alpha$  (either *MIN*, *MAX*, or *SUM*) for a sort  $\sigma$  is defined as a binary operation over the lifted sort  $?\langle\sigma\rangle$ , i.e.,  $\alpha: ?\langle\sigma\rangle \times ?\langle\sigma\rangle \rightarrow ?\langle\sigma\rangle$ . The data type  $?\langle\sigma\rangle$  is associated with the constructors *NotNull*:  $\sigma \rightarrow ?\langle\sigma\rangle$ , *Null*:  $?\langle\sigma\rangle$ , the accessor *Value*:  $?\langle\sigma\rangle \rightarrow \sigma$  (that maps any value *NotNull*( $a$ ) to  $a$ ), and the testers *IsNotNull*:  $?\langle\sigma\rangle \rightarrow \mathbb{B}$ , *IsNull*:  $?\langle\sigma\rangle \rightarrow \mathbb{B}$ . Regarding implementation, such data types are directly supported in the underlying solver Z3. (For *COUNT* the range sort is  $?\langle\mathbb{Z}\rangle$ .) In SQL, aggregation over an empty collection yields null and null elements in the collection are discarded, e.g., sum aggregation over an empty collection yields null. The definition of *MAX* (similarly for *MIN*) is:

$$MAX(x_0, x_1) \stackrel{\text{def}}{=} Ite(IsNull(x_0), x_1, Ite(IsNull(x_1), x_0, Ite(Value(x_0) > Value(x_1), x_0, x_1)))$$

The definition of *SUM* is:

$$SUM(x_0, x_1) \stackrel{\text{def}}{=} Ite(IsNull(x_0), x_1, Ite(IsNull(x_1), x_0, NotNull(Value(x_0) + Value(x_1))))$$

Let  $t: \rho$  be a term with a single free variable  $x_0: \sigma$ . Let  $a: \zeta$  be a term with a single free variable  $x_0: \sigma$ . Intuitively,  $\sigma$  is a tuple sort, both  $t$  and  $a$  are projections, and  $a$  corresponds to an aggregate parameter. For example (see the schema in Example 1 below)  $x_0$  is a row in the *Scores* table,  $t$  corresponds to the projection *Scores.StudentID*, and  $a$  corresponds to the projection *Scores.Points* in *MAX(Scores.points)*.

We declare the function symbol *Select* $^\alpha[t, a]: \mathbb{L}\langle\sigma\rangle \rightarrow \mathbb{L}\langle\mathbb{T}\langle\rho, \zeta\rangle\rangle$  and define a set of recursive axioms for it that for each element in the list collect the aggregated value with respect to  $a$  and then create a list of pairs that for each projection  $t$  provides that aggregated value. In order to define these axioms, arrays (mathematical maps) are used.

Given domain sort  $\sigma_1$  and range sort  $\sigma_2$ ,  $\mathbb{A}\langle\sigma_1, \sigma_2\rangle$  is the corresponding *array sort*. (In particular, the set sort  $\mathbb{S}\langle\sigma_1\rangle$  is synonymous with  $\mathbb{A}\langle\sigma_1, \mathbb{B}\rangle$ .) Declare

$$\begin{aligned} Select^\alpha[t, a] &: \mathbb{L}\langle\sigma\rangle \rightarrow \mathbb{L}\langle\mathbb{T}\langle\rho, \zeta\rangle\rangle, \\ Collect &: \mathbb{L}\langle\sigma\rangle \times \mathbb{A}\langle\rho, \zeta\rangle \times \mathbb{L}\langle\sigma\rangle \rightarrow \mathbb{L}\langle\mathbb{T}\langle\rho, \zeta\rangle\rangle, \\ List &: \mathbb{L}\langle\sigma\rangle \times \mathbb{A}\langle\rho, \zeta\rangle \rightarrow \mathbb{L}\langle\mathbb{T}\langle\rho, \zeta\rangle\rangle \end{aligned}$$

and define the following axioms, where *Read*:  $\mathbb{A}\langle\rho, \zeta\rangle \times \rho \rightarrow \zeta$  and *Store*:  $\mathbb{A}\langle\rho, \zeta\rangle \times \rho \times \zeta \rightarrow \mathbb{A}\langle\rho, \zeta\rangle$  are the standard built-in functions of the array theory. The empty array  $\varepsilon$  maps all elements of the domain

sort to the default value of the range sort. For lifted sorts the default value is null.

$$\begin{aligned}
\forall x (Select^\alpha[t, a](x) &= Collect(x, \varepsilon, x)) \\
\forall \bar{x} (Collect(cons(x_0, x_1), x_2, x_3) &= Collect(x_1, Store(x_2, t, \alpha(a, Read(x_2, t))), x_3)) \\
\forall \bar{x} (Collect(nil, x_0, x_1) &= List(x_1, x_0)) \\
\forall \bar{x} (List(cons(x_0, x_1), x_2) &= cons(T(t, Read(x_2, t)), List(x_1, x_2))) \\
\forall x (List(nil, x) &= nil)
\end{aligned}$$

In the current implementation, the above axioms are specialized to the case when the aggregate argument is required to be non null (for performance reasons), and the sort of  $a$  is not lifted. Although lifted sorts are avoided, this limitation requires special treatment of the cases when the collection is empty and implies that aggregates do not work with nullable column types.

## 4 From SQL to formulas

In this section we show how we translate an SQL query  $q$  into a set of axioms  $Th(q)$  that is suitable as an input soft theory to an SMT solver. The translation makes use of the list axioms discussed in Section 3. Although functional encodings of queries through comprehensions and combinators have been used earlier for compiler construction and query optimization (e.g. [14]), we are not aware of such encodings having been used for symbolic analysis or SMT solving. We illustrate the encodings here in order to make the paper self-contained. The concrete implementation with Z3 terms is very similar.

We omit full details of the translation and illustrate it through examples and templates, which should be adequate for understanding how the general case works. The focus is on the purely relational subset of SQL (without side-effects). We start by describing how tables are represented.

### 4.1 Tables and table constraints

Tables are represented by lists of rows where each row is a tuple. The sorts of the elements in the tuple are derived from the types of the corresponding columns that are given in the database schema. The currently supported column types in Qex are: `BigInt`, `Int`, `SmallInt`, `TinyInt`, `Bit`, and `Char`. The first four types are mapped to  $\mathbb{Z}$  (and associated with a corresponding range constraint, e.g., between 0 and 255 for `TinyInt`). `Bit` is mapped to  $\mathbb{B}$ . `Char` (that in SQL stands for a sequence of characters) is mapped to the *string sort* (or *word sort*)  $\mathbb{W} = \mathbb{L}\langle\mathbb{C}\rangle$ , where  $\mathbb{C}$  is the built-in sort of  $n$ -bitvectors for some fixed  $n$  that depends on the character range: UTF-16 ( $n = 16$ ), basic ASCII ( $n = 7$ ), extended ASCII ( $n = 8$ ).

The order of rows in a table is irrelevant regarding the evaluation semantics of queries. The number of times the same row occurs in a table is the *multiplicity* of the row. In general, duplicate rows are allowed in tables so the multiplicity may be more than one. However, in most cases input tables have primary keys that disallow duplicates. Tables may also be associated with other constraints such as foreign key constraints and restrictions on the values in the columns. In Qex, these constraints are translated into corresponding formulas on the list elements. The following example illustrates that aspect of Qex.

**Example 1.** Consider the following schema for a school database.

```

CREATE TABLE [School].[Scores]
(StudentID tinyint not null FOREIGN KEY REFERENCES Students(StudentNr),
 CourseID tinyint not null CHECK(1 <= CourseID and CourseID <= 100),

```

```
Points tinyint not null CHECK(Points <= 10),
PRIMARY KEY (StudentID, CourseID),
CHECK(NOT(1 <= CourseID and CourseID <= 10) or Points < 6));
```

```
CREATE TABLE [School].[Students]
(StudentNr tinyint not null PRIMARY KEY,
StudentName char(100) not null);
```

The (primary) key of the `Scores` table is the pair containing a student id and a course id and each row provides the number of points the student has received for the given course. The additional constraints are that the course ids go from 1 to 100, no course gives more than 10 points and courses 1 through 10 give a maximum of 5 points.

Qex declares the table variables  $Scores: \mathbb{L}\langle \mathbb{T}\langle \mathbb{Z}, \mathbb{Z}, \mathbb{Z} \rangle \rangle$  and  $Students: \mathbb{L}\langle \mathbb{T}\langle \mathbb{Z}, \mathbb{W} \rangle \rangle$ . There is a given bound  $k$  on the number of rows in each table. (In general there is a separate bound per table and the bounds are increased during model generation discussed in Section 5.) The following equalities are generated:

$$\begin{aligned} Scores &= cons(Scores_0, \dots, cons(Scores_{k-1}, nil)) \\ Students &= cons(Students_0, \dots, cons(Students_{k-1}, nil)) \end{aligned}$$

where  $Scores_i: \mathbb{T}\langle \mathbb{Z}, \mathbb{Z}, \mathbb{Z} \rangle$  and  $Students_i: \mathbb{T}\langle \mathbb{Z}, \mathbb{W} \rangle$  for  $i < k$ . For the primary key constraints, the following formulas are generated. The distinctness predicate and the projections functions  $\pi_i$  on tuples are built-in. We use  $t.i$  to abbreviate the term  $\pi_i(t)$ .

$$\begin{aligned} &Distinct(T(Scores_0.0, Scores_0.1), \dots, T(Scores_{k-1}.0, Scores_{k-1}.1)) \\ &Distinct(Students_0.0, \dots, Students_{k-1}.0) \end{aligned}$$

For expressing the foreign key constraint, Qex uses the built-in sets and the subset predicate:

$$\{Scores_i.0\}_{i < k} \subseteq \{Students_i.0\}_{i < k}$$

Currently, foreign key constraints are not supported over nullable types. The remaining constraints are conjunctions of check-constraints on individual rows, e.g.,

$$\bigwedge_{i < k} (\neg(1 \leq Scores_i.1 \wedge Scores_i.1 \leq 10) \vee Scores_i.2 < 6)$$

asserts that courses 1 through 10 give a maximum of 5 points. ⊠

## 4.2 Nullable values

If a column in a table is optional, it may contain “null” as a placeholder. Any column in SQL (other than a primary key column) is optional unless a `not null` type constraint is associated with the column type. Algebraic data types provide a convenient mechanism to represent optional values through *lifted sorts* as defined in Section 3.2.6.

When an SQL expression  $E$  is encoded as a term  $\llbracket E \rrbracket$ , it is assumed that  $E$  is *well-formed*: in the current implementation, operations using optional values are assumed to occur in a context where the value is known to be not null. SQL includes particular predicates `IS NULL` and `IS NOT NULL` for this purpose. In the translation the corresponding testers are used and the *Value* accessor is applied to cast the optional value to its underlying sort.

For example, assuming the column `Points` of the `Scores` table is declared `NOT NULL` (as in Example 1), the expression  $E = Points > 3$  is translated to  $Scores.2 > 3$ , but if `Points` is nullable, the expression  $E$  would have to occur in a context that is guarded by `Points IS NOT NULL`, e.g.,  $E = Points IS NOT NULL AND Points > 3$ , in which case  $\llbracket E \rrbracket$  is  $IsNotNull(Scores.2) \wedge Value(Scores.2) > 3$ .

Currently, well-formedness is not automatically detected and automatic support for such transformations is on the to-do-list. Regarding aggregates, the current implementation of Qex does not support aggregation over nullable types and a proper support for nullable values in combination of aggregates requires and adaptation of the corresponding axioms, which is yet another item on the to-do-list.

### 4.3 Formulas for queries

As the concrete input of queries, Qex uses a subset of the abstract syntax of the TSQL [1] grammar and the parser `TSq1100Parser` that is available in the VSTS'08 database edition. The currently supported constructs, some of which are also illustrated in the examples, are

- Selection, projection, group-by with having clause, inner join, nested queries.
- Aggregates: MIN, MAX, COUNT, SUM.
- Check constraints (composite), foreign and primary key constraints (composite).
- Arithmetic operations including negative numbers.
- Like-patterns and string length constraints.
- Restricted form of null support in table schemas.

We refer to the supported fragment by  $SQL^-$ . For dealing with null, the current translation does not fully support key constraints where values may be null or aggregates over columns where null is possible. Some of the corner cases require careful special handling in order to stay faithful to the semantics of SQL. Similarly, variable length string types and various character encodings are currently not supported. Although the underlying solver is capable of supporting full Unicode, the current experiments assume ASCII character encoding. The following constructs are currently not supported.

- Nested queries in from clauses. Correlated nested queries. Other join operations besides inner join.
- Set-type operands in where clauses, exists-expressions and in-expressions. Set operations such as union, intersection and difference.
- Order by.
- Store procedures.

Regarding the first two items, there is a plan to support most common cases. Order by clauses are viewed as postprocessing of the result and are currently not planned to be supported as part of model generation. Store procedures fall outside the scope of this paper, although there are future plans to look into symbolic execution of store procedures.

It is not feasible to fit the details of the translation from queries to formulas into the paper, instead, we look at a collection of representative samples that illustrate the core ideas behind the translation. In the samples, we reuse the schema from Example 1. We denote the term resulting from an  $SQL^-$  expression  $E$  by  $\llbracket E \rrbracket$ . The overall goal of the translation is summarized by the following proposition. Given a list  $l$  let  $\{\{l\}\}$  denote the corresponding multiset where the order of list elements is removed.

**Proposition 1.** *Let  $q$  be an  $SQL^-$  query using input table references  $X_i, i < n$ , let  $\psi$  be a formula expressing the input table constraints, and let  $\phi$  be a condition over the result  $Y$  of  $q$ . If  $Th(q) \wedge \psi \wedge \phi \wedge Y = \llbracket q \rrbracket$  has a model  $M$  then  $\{\{X_i^M\}\}, i < n$ , is a set of input tables satisfying  $\psi$  and the evaluation of  $q$  with respect to the input tables produces the result  $\{\{Y^M\}\}$  satisfying  $\phi$ .*



*Proof (sketch).* The complete proof uses induction over the structure of  $SQL^-$  expressions and is contingent upon a complete definition of  $SQL^-$  as well as a formal mapping of SQL types to the corresponding background sorts. For example, the select clause has the following abstract syntax in simplified form:

$$\begin{aligned} \textit{select\_clause} ::= & \text{SELECT } [\text{DISTINCT}] \textit{select\_list} \\ & \text{FROM } \textit{table\_src} [\text{WHERE } \textit{condition}] [\textit{group\_by\_having}] \end{aligned}$$

The translation of a select clause depends on whether grouping is used and whether aggregates occur in the select list. Suppose  $q$  is a simple select clause `SELECT  $L$  FROM  $T$  WHERE  $C$`  without aggregates. The expression `FROM  $T$  WHERE  $C$`  is translated to  $t = \textit{Filter}[\llbracket C \rrbracket](\llbracket T \rrbracket)$  that filters out all elements in the list  $\llbracket T \rrbracket$  that do not satisfy the condition  $\llbracket C \rrbracket$ . Note that this translation preserves the multiplicities of the elements in  $\llbracket T \rrbracket$  and is consistent with the multiset semantics. The translation  $\llbracket q \rrbracket$  of  $q$  is  $\textit{Map}[\llbracket L \rrbracket](t)$  where the elements of  $t$  are projected according to  $L$ . This translation also preserves the multiset semantics even if the projection  $\llbracket L \rrbracket$  is not injective, i.e., several occurrences of the same element may arise as a result of the map operation. Other  $SQL^-$  expressions are treated similarly.  $\square$

### 4.3.1 SELECT clauses

The main component of a query is a *select clause*. A select clause refers to a particular selection of columns from a given table by using a *select list*. The table is often a derived table, as the result of a join operation. Consider the query  $q$ :

```
SELECT StudentName, Points
FROM Students JOIN Scores ON Scores.StudentID = Students.StudentNr
WHERE Scores.CourseID = 10 AND Scores.Points > 0
```

The formula  $\llbracket q \rrbracket$  is:

$$\textit{Map}[T(x.0.1, x.1.2)](\textit{Filter}[x.1.1 = 10 \wedge x.1.2 > 0](\textit{Filter}[x.0.0 = x.1.0](\textit{Cross}(\textit{Students}, \textit{Scores}))))$$

where  $x: \mathbb{T}\langle \mathbb{T}\langle \mathbb{Z}, \mathbb{W} \rangle, \mathbb{T}\langle \mathbb{Z}, \mathbb{Z}, \mathbb{Z} \rangle \rangle$ . Such formulas get human-unreadable very quickly. During the process of creating  $\llbracket q \rrbracket$ , usually several list axioms are created. This set of axioms is referred to as  $\textit{Th}(q)$ . In particular, in this case  $\textit{Th}(q)$  includes the axioms for the map, filter, and cross product function symbols that occur in  $\llbracket q \rrbracket$ .

### 4.3.2 Aggregates

Aggregates are used to combine values from a group of rows in a table. The most common aggregates are MIN, MAX, SUM, and COUNT. For example, the following query  $q_1$  selects the maximum points from the Scores table.

```
SELECT MAX(Points) from Scores
```

Depending on the use of  $q_1$ , the translation  $\llbracket q_1 \rrbracket$  is either the singleton list:

$$\textit{cons}(T(\textit{Reduce}[\textit{Ite}(x_0.2 \geq x_1, x_0.2, x_1)](\textit{Scores}, \textit{MinValue}(\mathbb{Z}))), \textit{nil})$$

or just the *Reduce*[]-term:

$$\textit{Reduce}[\textit{Ite}(x_0.2 \geq x_1, x_0.2, x_1)](\textit{Scores}, \textit{MinValue}(\mathbb{Z}))$$

The first case applies if  $q_1$  is used as a top-level query, the second case applies if  $q_1$  is used as a *subquery expression*. The second case applies in the following query  $q_2$  that also uses the MAX aggregate in the top level select list in combination with GROUP BY that eliminates duplicates from the resulting table:

```
SELECT StudentID, MAX(Points) FROM Scores GROUP BY StudentID
HAVING MAX(Points) = (SELECT MAX(Points) from Scores)
```

The query  $q_2$  selects all students that have the most points at some course. The translation of  $[[q_2]]$  is as follows where the  $Filter[]$  application corresponds to the HAVING clause that is applied to the result of the grouping.

$$Filter[x.1 = [[q_1]]](RemoveDuplicates(Select^{MAX}[x_0.0, x_0.3](Scores)))$$

### 4.3.3 LIKE-patterns

Like-patterns are particular regular expressions that can be used as constraints on strings. A like-pattern  $r$  is converted into a *symbolic finite automaton* [27] (SFA)  $A_r$  that is similar to a classical finite automaton except that moves are labeled by formulas denoting *sets* of characters rather than single characters. The full expressiveness of patterns  $r$  that is currently supported by the conversion  $A_r$  is that of .NET regexes (except for anchors  $\backslash G$ ,  $\backslash b$ ,  $\backslash B$ , named groups, lookahead, lookbehind, as-few-times-as-possible quantifiers, backreferences, conditional alternation, and substitution).

The automaton  $A_r$  is translated into a theory  $Th(A_r)$ . The theory describes the acceptance condition for words in  $L(A_r)$ . In particular,  $Th(A_r)$  defines a predicate

$$Acc^{A_r} : \mathbb{W} \times \mathbb{N} \rightarrow \mathbb{B},$$

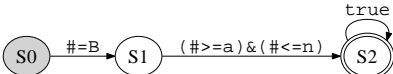
where  $\mathbb{N}$  is an algebraic datatype for *unary natural numbers* with the constructors  $\bar{0} : \mathbb{N}$  and  $\mathbf{s} : \mathbb{N} \rightarrow \mathbb{N}$ . We write  $\bar{k} + \bar{1}$  for  $\mathbf{s}(\bar{k})$ . Intuitively,  $Acc^{A_r}(t, \bar{k})$  expresses that  $t$  is a word of at most  $k$  characters that matches the pattern  $r$ . We use the following property of the theory of  $A_r$  [27, Theorem 1]:

**Proposition 2.** *Let  $t$  be a closed term of sort  $\mathbb{W}$ ,  $k$  a nonnegative integer, and  $M$  a model of  $Th(A_r)$ . Then  $M \models Acc^{A_r}(t, \bar{k})$  iff  $t^M \in L(A_r)$  and  $|t^M| \leq k$ .*

In column type declarations of SQL database schemas, a maximum string length is associated with the `char` type (default being 1), e.g., the type `char(100)` of a column allows strings containing at most 100 characters. In the formula  $Acc^{A_r}(t, \bar{k})$ , where  $t$  refers to a column whose values are strings,  $k$  is the maximum length of the strings in that column.

**Example 2.** Consider the query  $q$  that selects students whose name starts with the letter B followed by any letter between a and n followed by 0 or more additional characters:

```
SELECT StudentName FROM Students WHERE StudentName like "B[a-n]%"
```

The SFA  $A$  for "B[a-n]%" is  where  $\#$  is a free variable of sort  $\mathbb{C}$  and each symbolic move  $(i, \varphi[\#], j)$  denotes the set of transitions  $\{(i, a, j) \mid a \in \mathcal{U}^{\mathbb{C}}, \models \varphi[a]\}$ . For each state  $S0, S1, S2$  of  $A$  there are two axioms in  $Th(A)$ , one for length bound = 0 and the other one for length bound  $> 0$ :

$$\begin{aligned} S0 : \quad & \forall x (Acc(x, \bar{0}) \Leftrightarrow false) & \forall xy (Acc(x, \mathbf{s}(y)) \Leftrightarrow x \neq nil \wedge hd(x) = B \wedge Acc_1(tl(x), y)) \\ S1 : \quad & \forall x (Acc_1(x, \bar{0}) \Leftrightarrow false) & \forall xy (Acc_1(x, \mathbf{s}(y)) \Leftrightarrow x \neq nil \wedge hd(x) \geq \mathbf{a} \wedge hd(x) \leq \mathbf{n} \wedge Acc_2(tl(x), y)) \\ S2 : \quad & \forall x (Acc_2(x, \bar{0}) \Leftrightarrow x = nil) & \forall xy (Acc_2(x, \mathbf{s}(y)) \Leftrightarrow x = nil \vee (x \neq nil \wedge Acc_2(tl(x), y))) \end{aligned}$$

The term  $[[q]]$  is  $Map[T(x_0.1)](Filter[Acc(x_0.1, \bar{100})](Students))$ . Note that  $Th(A) \subseteq Th(q)$ .  $\square$

The automata based approach opens up several transformation techniques that can be performed in the process of encoding queries and theories of queries that involve like-patterns. These upfront transformations can greatly simplify the formulas. We illustrate this with an example involving the use of *product* of SFAs. The following proposition follows directly from the product definition (see [27]).

**Proposition 3.** *Let  $A$  and  $B$  be SFAs, then  $L(A \otimes B) = L(A) \cap L(B)$ .*

**Example 3.** Consider the following query  $q_{\text{LIKE}}$  with  $n + 1$  occurrences of “\_” in the first like-pattern and  $n$  occurrences of “\_” in the second like-pattern:

```
SELECT StudentName FROM Students
WHERE StudentName like "%a_-----" AND StudentName like "%b_-----"
```

The first like-pattern corresponds to the regex  $r_1 = . * a . \{n+1\}$  and the second like-pattern corresponds to the regex  $r_2 = . * b . \{n\}$ . The query is essentially an intersection constraint of  $r_1$  and  $r_2$ . In a direct encoding of  $q_{\text{LIKE}}$ ,  $Th(q_{\text{LIKE}})$  includes both the axioms for  $A_{r_1}$  as well as  $A_{r_2}$ . Rather than using  $A_{r_1}$  and  $A_{r_2}$  separately, the product  $A_{r_1} \otimes A_{r_2}$  of  $A_{r_1}$  and  $A_{r_2}$  can be used together with the theory  $Th(A_{r_1} \otimes A_{r_2})$  instead of  $Th(A_{r_1}) \cup Th(A_{r_2})$ . Thus, with *product encoding*,

$$\llbracket q_{\text{LIKE}} \rrbracket = \text{Map}[T(x_0.1)](\text{Filter}[\text{Acc}^{A_{r_1} \otimes A_{r_2}}(x_0.1, \overline{100})](\text{Students}))$$

and with *direct encoding*,

$$\llbracket q_{\text{LIKE}} \rrbracket = \text{Map}[T(x_0.1)](\text{Filter}[\text{Acc}^{A_{r_1}}(x_0.1, \overline{100}) \wedge \text{Acc}^{A_{r_2}}(x_0.1, \overline{100})](\text{Students}))$$

The gain in performance is discussed in Section 6. ⊠

Note that correctness of the transformation illustrated in Example 3 follows from Propositions 2 and 3.

## 5 Implementation

Qex uses the SMT solver Z3 [30, 10]. Interaction with Z3 is implemented through its programmatic API rather than using a textual format, such as the smt-lib format [24]. The main reasons for working with the API are: access to built-in data types; model generation; working within a given context. The first point is fundamental, since algebraic data types are central to the whole approach and are not part of the smt-lib standard.

Besides allowing to check satisfiability, perhaps the most important feature exposed by some SMT solvers (including Z3) for the purposes of test input generation is *generating a model* as a witness of the satisfiability check, i.e., a mapping of the uninterpreted function symbols to their interpretations. Z3 has a separate method for satisfiability checking with model generation. This code snippet illustrates the use of that functionality:

```
Model m;
z3.AssertCnstr(f);
LBool sat = z3.CheckAndGetModel(out m);
Term v = m.Eval(s); ...
```

A *context* includes declarations for a set of symbols, and assertions for a set of formulas. A context is essentially a layering mechanism for signature expansions with related constraints. There is a *current context* and a backtrack stack of previous contexts. Contexts can be saved through *pushing* and restored through *popping*. When a satisfiability check is performed in a given context, the context may become inconsistent. Qex uses contexts during table generation and in SFA algorithms during theory generation for like-patterns.

## 5.1 Incremental table generation

Let  $q$  be a fixed query and assume that  $X_1, \dots, X_n$  are the input table variables. Assume  $\llbracket q \rrbracket : \sigma$  and let  $\varphi[Y]$  be a formula with the free variable  $Y : \sigma$ . Intuitively,  $\varphi$  is a *test condition* on the result of the query, e.g.  $Y \neq nil$ . The following *basic table generation procedure* describes the input table generation for  $q$  and  $\varphi$ .

1. Assert  $Th(q)$ , i.e. add the axioms of  $q$  to the current context.
2. Let  $\vec{k} = (k_1, \dots, k_n) = (1, \dots, 1)$  be the initial sizes of the input tables. Repeat the following until a model  $M$  is found or a timeout occurs.
  - (a) Push the current context, i.e., create a backtrack point.
  - (b) Create constraints for  $X_1, \dots, X_n$  using  $\vec{k}$  to fix the table sizes.
  - (c) Assert  $\varphi[\llbracket q \rrbracket]$
  - (d) Check and get the model  $M$ . If the check fails, increase  $\vec{k}$  systematically (e.g., by using a variation of Cantor’s enumeration of rationals), and pop the context.
3. Get the values of  $X_1, \dots, X_n$  in  $M$ .

There are several possible variations of the basic procedure. The table constraints can be updated incrementally when the table sizes are increased. The table constraints can also be created for *upper* bounds rather than exact bounds on the table sizes. One way to do so is as follows:

$$table = cons(row1, rest1) \wedge (rest1 = nil \vee (rest1 = cons(row2, rest2) \wedge \dots))$$

The size of the overall resulting formula is always polynomial in the size of the original query and  $\vec{k}$ . In practice, Qex uses bounds on  $\vec{k}$  and overall time constraints to guarantee termination, as deciding the satisfiability of queries is undecidable in general [9].

## 5.2 Symmetry breaking formulas

The translation of a query  $q$  into a formula  $\llbracket q \rrbracket$  together with  $Th(q)$  and the additional table constraints looks very much like a “functional program with constraints”. This intuition is correct as far as the logical meaning of the translation is concerned. There are, however, no mechanisms to control the evaluation order of patterns (such as, “outermost first”) and no notion of term orderings. The search space for  $\llbracket q \rrbracket$  is typically vast.

Recall that although Qex uses lists to encode tables, the order of rows is not relevant according to the SQL semantics. We can therefore assert predicates that constrain the input tables to be *ordered* (thus eliminating all symmetrical models where the ordering does not hold). Consider a table  $cons(row_0, cons(row_1, \dots, cons(row_n, nil)))$  of sort  $\mathbb{L}(\sigma)$ . Define a lexicographic order predicate  $\preceq : \sigma \times \sigma \rightarrow \mathbb{B}$ . The definition of  $\preceq$  on integers is just the built-in order  $\leq$ , similarly for bitvectors. For tuples, it is the standard lexicographic order defined in terms of the orders of the respective element sorts. For strings (lists of bitvectors) the order predicate can be defined using recursion over lists. Assert the *symmetry breaking formula*

$$\bigwedge_{i < n-1} row_i \preceq row_{i+1}$$

In some situations the symmetry breaking formula can be strengthened. For example, when the table has a primary key then the formula can be strengthened by using the strict order  $\prec$  instead of  $\preceq$ . Moreover, if all of the columns are part of the primary key then the primary key constraint itself becomes redundant.

Table 2: Sample queries.

#	Query	$t$ [ms]	$k$
1	<pre> DECLARE @x as tinyint; SELECT Scores.StudentID, SUM(Scores.Points) FROM Scores WHERE Scores.Points &gt; 2 GROUP BY Scores.StudentID HAVING SUM(Scores.Points) &gt;= @x AND @x &gt; 5 </pre>	20	1
2	<pre> SELECT Scores.StudentID, MAX(Scores.Points) FROM Scores GROUP BY Scores.StudentID HAVING MAX(Scores.Points) = (SELECT MAX(Scores.Points) FROM Scores) </pre>	20	1
3	<pre> DECLARE @x as tinyint; SELECT COUNT(S.StudentName) FROM Students as S WHERE S.StudentName LIKE "%Mar[gkc]us%" AND S.StudentNr &gt; @x HAVING COUNT(S.StudentName) &gt; @x AND @x &gt; 2 </pre>	1300	4
4	<pre> DECLARE @x as tinyint; SELECT Students.StudentName, SUM(Points) FROM Scores JOIN Students ON Scores.StudentID = Students.StudentNr WHERE Scores.Points &gt; 2 AND Students.StudentName LIKE "John%" GROUP BY Students.StudentName HAVING SUM(Points) &gt;= @x AND @x &gt; 15 </pre>	200	2
5	<pre> SELECT Students.StudentName, Scores.Points FROM Scores JOIN Students ON Scores.StudentID = Students.StudentNr WHERE Scores.CourseID = 10 AND Scores.Points &gt; 0 </pre>	30	1
6	<pre> SELECT Students.StudentName, Courses.CourseName, Scores.Points FROM Scores JOIN Students ON Scores.StudentID = Students.StudentNr JOIN Courses ON Courses.CourseNr = Scores.CourseID WHERE Scores.Points &gt; 2 AND Students.StudentName LIKE "bob%" AND Courses.CourseName LIKE "AI" </pre>	80	1

## 6 Experiments

We provide some performance evaluation results of Qex on a collection of sample queries.<sup>1</sup> In the first set of experiments we look at the performance of the basic table generation procedure. In these experiments we use the same bound  $k$  for both tables. The test condition used here is that the result is nonempty. Table 2 summarizes the overall time  $t$  (in ms) for each query  $q$ , which includes the parsing time, the generation time of  $Th(q)$ , and the model generation time. (Note that query #3 is a valid SQL query without a group-by clause.) The column  $k$  shows the number of rows generated for the input tables. Some of the queries include parameters, indicated with @, the values of parameters are also generated. (The actual data that was generated is not shown here.) We reuse the schema introduced in Example 1. The last query uses an additional table called Courses with the schema:

```

CREATE TABLE [School].Courses
(CourseNr tinyint not null PRIMARY KEY, CourseName char(15) not null);

```

Using symmetry breaking over lists did not improve the performance for these examples. In some cases it had the opposite effect, e.g., for query #3 the time increase is 30%. Although symmetry breaking was crucial for the examples in [28], here the benefits are unclear. If we consider the test condition that the result has 4 rows, and also that the input tables all have 9 rows then, for query #6 the total time to generate the three input tables is 75s without symmetry breaking and 45s with symmetry breaking.

<sup>1</sup>The experiments were run on a Lenovo T61 laptop with Intel dual core T7500 2.2GHz processor.

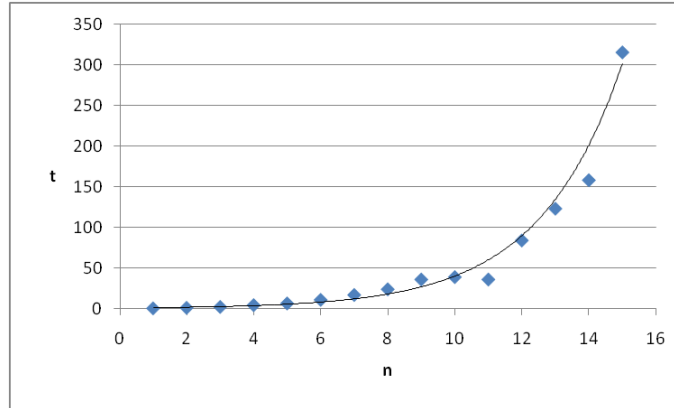


Figure 1: Exploration times (sec) for query #3 in Table 2 when the constant 2 is replaced with  $n$  for  $n = 1, \dots, 15$ .

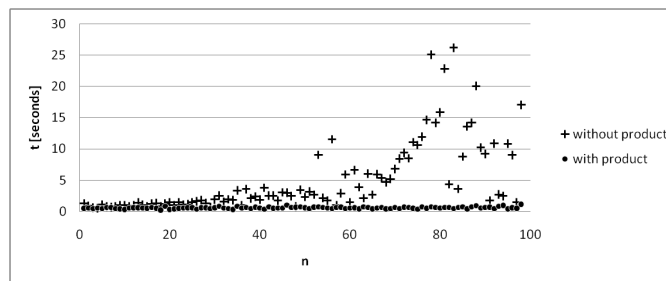


Figure 2: Exploration times (sec) for query  $q_{\text{LIKE}}$  without (scattered crosses) and with (solid line of dots at the bottom) product construction for  $n = 1, \dots, 98$ .

However, in general it seems that the ordering constraints on strings are expensive. At this point we do not have enough experience to draw clear conclusions when it pays off to use them.

The total size of the query seems to have very little effect on the time  $t$ . The key factor is the use of *aggregates* and the constraints they introduce that cause the input tables to grow, thus, causing backtracking during model generation, that is clearly seen for query #3. Consider the following experiment. Take query #3 and replace the constant 2 in it with the constant  $n$  for  $n = 1, \dots, 15$ . Figure 1 shows the time  $t$  in seconds as a function of  $n$ ;  $k$  is always  $n + 2$ .

Given a query  $q$ , several optimizations or transformations can be performed on the term  $\llbracket q \rrbracket$  as well as the set of axioms  $Th(q)$  prior to asserting them to the solver. Figure 2 shows a drastic decrease in model generation time for  $q_{\text{LIKE}}$  from Example 3 in Qex when the product construction is used. By performing localized SMT solver queries during product construction of SFAs, the size of the resulting automata can often dramatically decrease. We have experimented with a few special cases of this nature, but have not systematically applied such transformations or other transformations such as combining several consecutive filters as a single filter.

We also reevaluated the performance of Qex on the benchmarks reported in [28, Table 1] that use a different sample database schema (where strings do not occur). In all of the cases the performance improvement was between 10x and 100x. As we suspected, the eager expansion time reported as  $t_{\text{exp}}$  in [28], that was by an order of magnitude larger than the model generation time  $t_{23}$ , is avoided completely in the new approach. The initial cost of creating  $\llbracket q \rrbracket$  is negligible, since the size of  $\llbracket q \rrbracket$  is polynomial in

the size of  $q$  in theory, and close to linear in practice. The added overhead during model generation due to the use of axioms only marginally increased the model generation time  $t_{z3}$ .

The final example illustrates an application of the tool on normal form analysis of schemas.

**Example 4.** Consider the following additional schema of a table Learning.

```
CREATE TABLE [School].Learning
(Student TINYINT NOT NULL, Course TINYINT NOT NULL, Teacher TINYINT NOT NULL,
PRIMARY KEY (Student, Course),
UNIQUE (Student, Teacher));
```

It is easy to see that the table satisfies 3NF (3rd normal form) since all attributes are prime (belong to a candidate key of Learning). Suppose that there is a functional dependency  $\text{Teacher} \rightarrow \text{Course}$ . One can show that the table does not satisfy BCNF (Boyce-Codd normal form, it is a slightly stronger version of 3NF), where for each functional dependency  $X \rightarrow Y$ ,  $X$  must be a superkey (i.e. a candidate key or a superset of a candidate key). One can show that Teacher is not a superkey of the Learning table by showing that the following query can yield a nonempty answer:

```
SELECT X.Teacher
FROM Learning AS X JOIN Learning AS Y ON (X.Teacher = Y.Teacher)
WHERE X.Course < Y.Course; (2)
```

The basic table generation procedure for (2) provides the following solution for Learning:

Student	Course	Teacher
0	69	133
1	70	133

The following query can be used to serve the same purpose:

```
SELECT Teacher, COUNT(Course)
FROM Learning
GROUP BY Teacher HAVING COUNT(Course) > 1; (3)
```

For both queries (2) and (3) the total execution time is around 20 milliseconds: we repeated the experiment for (2) 100 times with total execution time of 2 seconds, and we repeated the experiment with (3) 100 times as well, with total execution time of 2 seconds also. The actual model generation time for a single run was around 10 milliseconds for both queries, the rest of the time was due to the overhead of the startup, file handling and parsing. However, by changing query (3) slightly, by replacing  $\text{COUNT}(\text{Course}) > 1$  by  $\text{COUNT}(\text{Course}) > n$ , for  $n > 1$ , one can detect exponential increase in model generation time: for  $n = 4; 5; 6$  the experiment took 0.1; 0.2; 0.4 seconds.  $\square$

## 7 Related work

The first prototype of Qex was introduced in [28]. The current paper presents a continuation of the Qex project [23], and a redesign of the encoding of queries into formulas based on a lazy axiomatic approach that was briefly mentioned in [28] but required support for algebraic data types in the underlying solver. Moreover, Qex now also supports a substantially larger fragment of SQL (such as subquery expressions) and like-patterns on strings, as discussed above.

Deciding satisfiability of SQL queries requires a formal semantics. While we give meaning to SQL queries by an embedding into the theory of an SMT solver, there are other approaches, e.g., defining

the semantics in the Extended Three Valued Predicate Calculus [19], or using bags as a foundation [7]. Satisfiability of queries is also related to logic-based approaches to semantic query optimization [5]. The general problem of satisfiability of SQL queries is undecidable and computationally hard for very restricted fragments, e.g., deciding if a query has a nonempty answer is NEXP-hard for nonrecursive range-restricted queries [9].

Several research efforts have considered formal analysis and verification of aspects of database systems, usually employing a possibly interactive theorem prover. For example, one system [25] checks whether a transaction is guaranteed to maintain integrity constraints in a relational database; the system is based on Boyer and Moore-style theorem proving [4].

There are many existing approaches to generate database tables as test inputs. Most approaches create data in an ad-hoc fashion. Only few consider a target query. Tsai et.al. present an approach for test input generation for relational algebra queries [26]. They do not use lists to represent tables. They propose a translation of queries to a set of systems of linear inequalities, for which they implemented an ad-hoc solving framework which compares favorably to random guessing of solutions. A practical system for testing database transactions is AGENDA [12]. It generates test inputs satisfying a database schema by combining user-provided data, and it supports checking of complex integrity constraints by breaking them into simpler constraints that can be enforced by the database. While this system does not employ a constraint solver, it has been recently refined with the TGQG [6] algorithm: Based on given SQL statements, it generates test generation queries; execution of these queries against a user-provided set of data groups yields test inputs which cover desired properties of the given SQL statements.

Some recent approaches to test input generation for databases employ automated reasoning. The relational logic solver Alloy [16, 17] has been used by Khalek et.al. [18] to generate input data for database queries. Their implementation supports a subset of SQL with a simplified syntax. In queries, they can reason about relational operations on integers, equality operations on strings, and logical operations, but not about nullable values, or grouping with aggregates such as SUM; they also do not reason about duplicates in the query results. QAGen [3] is another approach to query-solving. It first processes a query in an ad-hoc-way, which requires numerous user-provided “knob” settings as additional inputs. From the query, a propositional logic formula is generated, which is then decided by the Cogent [8] solver to generate the test inputs. In [2] a model-checking based approach, called Reverse Query Processing, is introduced that, given a query and a result table as input, returns a possible database instance that could have produced that result for that query, the approach uses reverse relational algebra. In [29] an intentional approach is presented in which the database states required for testing are specified as constrained queries using a domain specific language. Recently, test input generation of queries has been combined with test input generation of programs that contain embedded queries in the program text [13], using ad-hoc heuristic solvers for some of the arising constraints from the program and the queries.

Generating sample input data for databases is related to generating sample data for dataflow programs, the work in [20] discusses input data generation for Pig Latin [21], developed at Yahoo! Research, that is a query language in between SQL and the mapreduce [11] programming model. The approach in [20] focuses on certain core aspects of Pig Latin that can also handle aggregation through GROUP and TRANSFORM constructs of the language. The algorithm in [20] does not use off-the-shelf tools or symbolic analysis techniques but is a stand-alone multi-pass dataflow analysis algorithm. It is unclear as to how the approach can be combined with additional constraints, for example arithmetical constraints or string constraints in form of regular patterns.



## 8 Conclusion and future work

The current implementation of the Qex project is still in its early stages, but we were highly encouraged by the performance improvements when switching to the lazy approach and reducing the need for nonlinear constraints through a different representation of tables. There are many more possible optimizations that can be performed as a preprocessing step on formulas generated by Qex, before asserting them to the SMT solver. One such optimization, using automata theory, was illustrated in Example 3 and Figure 2 when multiple like-patterns occur in a query. Systematic preprocessing can also often reveal that a query is trivially false, independent of the size of input tables, e.g., if an ‘\_’ is missed in the first like-pattern in Example 3 then the product automaton would be empty.

For practical usage in an industrial context, where SQL queries are usually embedded in other programs or in store procedures, we are looking at integrating Qex in Pex [22]. For efficient support for regex constraints in Pex, integration of Rex [27] is a first step in that integration.

It is also possible to apply a translation similar to the one described in the paper to LINQ queries, although, unlike in SQL, the semantics of LINQ queries depends on the order of the rows in the tables. This fits well with the list representation of tables but imposes some limitations on the use of certain optimizations (such as the use of symmetry breaking formulas).

A practical limitation of Qex is if queries use multiple joins and aggregates and the input tables need to contain a high number of rows in order to satisfy the test condition. Another limitation is the use nonlinear constraints over unbounded integers, in particular multiplication, that has currently only limited support in Z3. We consider using bitvectors instead. Despite these limitations, the mere size of queries does not seem to be a concern, neither the size of  $Th(q)$  for a given query  $q$ . The size of  $Th(q)$  may easily be in hundreds, in particular when several like-patterns are used, where the number of axioms is proportional to the size of the finite automaton accepting the pattern.

## Acknowledgements

We thank Nikolaj Bjørner for the continuous support with Z3 and for suggesting the idea of symmetry breaking formulas. We also thank the reviewers for many useful comments and for pointing out some related work that we were not aware of. One reviewer provided very detailed and constructive comments, up to an extent that could under different circumstances warrant coauthorship, in particular, suggested to formulate Proposition 1, provided a more accurate handling of null, corrected (essentially rewrote) the axioms in Section 3.2.6 (the original description was in an appendix, and besides having several typos, could not handle null), and provided the schema normalization Example 4 as an intuitive and different application of the techniques presented in the paper.

## References

- [1] SELECT (T-SQL). <http://msdn.microsoft.com/en-us/library/ms189499.aspx>.
- [2] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE 2007)*, pages 506–515. IEEE, 2007.
- [3] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. Qagen: generating query-aware test databases. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 341–352, New York, NY, USA, 2007. ACM.
- [4] R. S. Boyer and J. S. Moore. *A computational logic handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.
- [5] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15(2):162–207, 1990.

- [6] D. Chays, J. Shahid, and P. G. Frankl. Query-based test generation for database applications. In *Proceedings of the 1st International Workshop on Testing Database Systems (DBTest'08)*, pages 1–6, New York, NY, USA, 2008. ACM.
- [7] H. R. Chinaei. An ordered bag semantics of SQL. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 2007.
- [8] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *Proceedings of CAV 2005, volume 3576 of Lecture Notes in Computer Science*, pages 296–300. Springer, 2005.
- [9] E. Dantsin and A. Voronkov. Complexity of query answering in logic databases with complex values. In *Proceedings of the 4th International Symposium on Logical Foundations of Computer Science (LFCS'97)*, pages 56–66, London, UK, 1997. Springer.
- [10] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'08)*, LNCS. Springer, 2008.
- [11] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.
- [12] Y. Deng, P. Frankl, and D. Chays. Testing database transactions with AGENDA. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 78–87, New York, NY, USA, 2005. ACM.
- [13] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 151–162. ACM, 2007.
- [14] T. Grust and M. H. Scholl. How to comprehend queries functionally. *Journal of Intelligent Information Systems*, 12(2-3):191–218, 1999.
- [15] W. Hodges. *Model theory*. Cambridge Univ. Press, 1995.
- [16] D. Jackson. Automating first-order relational logic. *SIGSOFT Softw. Eng. Notes*, 25(6):130–139, 2000.
- [17] D. Jackson. *Software Abstractions*. MIT Press, 2006.
- [18] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. Query-aware test generation using a relational constraint solver. In *ASE*, pages 238–247, 2008.
- [19] M. Negri, G. Pelagatti, and L. Sbatella. Formal semantics of SQL queries. *ACM Transactions on Database Systems*, 17(3):513–534, September 1991.
- [20] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *SIGMOD*, pages 245–256. ACM, 2009.
- [21] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110. ACM, 2008.
- [22] Pex. <http://research.microsoft.com/projects/pex>.
- [23] Qex. <http://research.microsoft.com/projects/qex>.
- [24] S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at [www.SMT-LIB.org](http://www.SMT-LIB.org).
- [25] T. Sheard and D. Stemple. Automatic verification of database transaction safety. *ACM Trans. Database Syst.*, 14(3):322–368, 1989.
- [26] W. T. Tsai, D. Volovik, and T. F. Keefe. Automated test case generation for programs specified by relational algebra queries. *IEEE Trans. Softw. Eng.*, 16(3):316–324, 1990.
- [27] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In A. Cavalli and S. Ghosh, editors, *Third International Conference on Software Testing, Verification and Validation (ICST 2010)*, Paris, France, April 2010. IEEE.
- [28] M. Veanes, P. Grigorenko, P. de Halleux, and N. Tillmann. Symbolic query exploration. In K. Breitman and A. Cavalcanti, editors, *ICFEM'09*, volume 5885 of LNCS, pages 49–68. Springer, 2009.
- [29] D. Willmor and S. M. Embury. An intensional approach to the specification of test cases for database applications. *28th International Conference on Software Engineering*, pages 102–111, 2006.
- [30] Z3. <http://research.microsoft.com/projects/z3>.