Design and Implementation of Reflective SQL

(Extended Abstract)

Mehmet M. Dalkilic* Manoj Jain* Dirk Van Gucht* Anurag Mendhekar†

Abstract

One weakness of SQL has been its inability to express certain classes of queries, e.g., iterative queries and database schema-independent queries. Fixes to this problem generally involve embedding SQL in a more expressive language. Reflection—the ability of a language to encode arbitrary programs and evaluate these encodings during execution—can provide a seamless, natural *solution*. In database query languages reflection has been generally confined to a procedural setting, where one paper in particular, Van den Bussche *et al*, has given us much of our inspiration; however, exactly how to pragmatically incorporate reflection into a query language like SQL has remained elusive. This paper presents the design and implementation of a *Reflective* SQL (RSQL): an extension of SQL wherein programs *themselves* can create (i.e., *reify*), *manipulate* and *evaluate* programs. Ordinary SQL tables serve to encode these programs; these tables look and feel much like the programs they encode and, furthermore, retain SQL's declarative nature. The implementation itself is comprised of an RSQL interface and a relational database server. This work establishes, in a practical setting, a means of enhancing SQL, opening new avenues to solutions of database problems that have been hitherto inaccessible through SQL alone.

*Computer Science Dept., Indiana University, Bloomington, IN 47405, USA. Email: {dalkilic,mjain,vgucht}@cs.indiana.edu
†Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA. Email: anurag@parc.xerox.com

1 Introduction

The relational data model with its query languages (relational algebra and relational calculus) introduced by Codd [3, 4] has been extensively used both as the theoretical and practical basis for designing commercial database systems. Significant properties of this model and its query languages include simplicity of use, relative ease of implementation, closure, and limited expressive power — the latter translating into a better use of shared resources. This limited expressiveness, however, has also been one of the major shortcomings of the relational model: there are certain classes of reasonable queries, e.g., iterative queries and database schema-independent queries, that simply cannot be formulated in the language.

Increasing the expressive power of the relational model and its query languages has been widely studied by both theoreticians and practitioners alike. Among these are (1) embedding query languages in a general purpose programming language; (2) adding various constructs such as transitive closure, fixpoint operators, and iteration to the query languages [1, 2, 5, 19]; (3) defining new models and languages that can express database schema-independent queries by allowing table and column names themselves to be treated as data [9, 10, 7].

These are *palliative* solutions, however, in that they target themselves to symptomatic weaknesses in the relational model: though the symptom is cured, either some other problems remain or, worse, they relinquish other properties of the relational model. For example, approach (1) suffers from the query language becoming *too* expressive, thus severely limiting the opportunities for optimization. Further, the relational abstraction is lost at the interface, where relations are mapped into disparate data structures, e.g., lists of records, violating the closure property. Approach (2), although retains the closure property, it cannot express schema-independent queries. Lastly, in approach (3) how to integrate the new models and query languages with a relational system is unclear.

A novel approach that effectively addresses inexpressiveness, schema-independence, etc., studied by Van den Bussche, Van Gucht, and Vossen [18], involves extending the query language with *reflection* mechanisms [12, 13, 14]: *reification*, the association of data with the meaning of a program, and *evaluation*, the association of the meaning of a program with data. Essentially, adding reflection to a programming language enables that language to encode arbitrary programs as data, manipulate these encodings, and evaluate these encodings during the execution of a program. In our paper we show how these reflection mechanisms allow a certain representation of queries within the framework of SQL itself and allow going back and forth between queries and their representation. Moreover, the representation of queries is chosen so that it may be manipulated without introducing new data structures to SQL.

In the field of databases, storing programs as data was initially investigated by Stonebraker *et al* [15, 16] where they proposed a system that allowed programs to be stored as strings in tuple components and executed dynamically. Van den Bussche *et al* [18] took this one step further by not only encoding programs as data, but also allowing *manipulation* of these encodings before evaluation. The true power of reflection comes from this manipulation of the encodings – that gives a program the ability to examine its own state and modify its own behavior depending upon this state. In [18], Van den Bussche *et al* presented a *reflective* relational algebra $\mathcal{RA}$ that attained reflection by (1) establishing a means of encoding relational algebra programs as relations and (2) adding an operator to evaluate these encodings. They observed that the relational algebra expressions, represented as *query trees*, could be naturally encoded as relations. These relations (called *program relations*) would then be manipulated in just the same way as the relations containing ordinary data (*ordinary relations*), and, furthermore, both these kinds of relations could mix freely in relational algebra computations. In order to store arbitrary query trees they added a mechanism for creating new data elements (the relational algebra does not provide such a mechanism in its pure sense), which is important for the generation of intermediate nodes for query trees. In a typical $\mathcal{RA}$-program, one *dynamically* constructs relational algebra queries with reification tools and subsequently evaluates these queries. The crucial observation in [18] was that, while an $\mathcal{RA}$-program itself is static, the relational algebra queries it constructs are dynamic in that they utilize, i.e., take as input, the current state of the database.

Adding reflection not only gives the ability to express the certain class of queries mentioned above, but also has numerous applications, such as procedural data, query optimization, polymorphism, inheritance, etc., (see [18, 6]). An important point to note is that extending a relational query language with reflection mechanisms is upwardly compatible to relational query languages with iterative constructs. Moreover, this approach does not introduce any new data structures, as we have mentioned, and it maintains the desired closure properties of the relational query languages.

Yet, what $\mathcal{RA}$ gained in its expressiveness over relational algebra, it lost in its use pragmatically; the encodings are neither easily managed nor handled, and discerning the relational expression from its encoding is generally quite difficult. In this paper, we present an reflective extension of SQL (called RSQL) whereby the programs *themselves* can create, manipulate, and execute programs. Ordinary SQL tables serve to encode these programs; these tables look and feel much like the programs they encode and, furthermore, retain SQL's declarative nature. The implementation itself is comprised of an RSQL interface and a relational database server. This work establishes, in a practical setting, a means of enhancing SQL, thereby opening new avenues to solutions of database problems that have been hitherto inaccessible through SQL alone.

This paper is organized as follows: Section 2 discusses how arbitrary SQL programs are encoded into tables; Section 3 presents in detail two, new operators *reify* and *eval* that encode and evaluate encodings by way of examples like reachability and schema-independent queries; Section 4 describes the RSQL system architecture; Section 5 closes with a summary and various points concerning RSQL.

2 Reflective SQL

Adding reflection to SQL requires an encoding of arbitrary SQL statements into tables and two additional operators that encode statements and evaluate encodings. In this section we focus upon the encoding

Program	::=	*Statement* ; [*Program*]
Statement	::=	*Query* | *Select-Into* | *Insert* | *Delete* | *Update*
Query	::=	*Select From* [*Where*] [*Order-by*] [*Union*]
Select-Into	::=	*Select* [*Into*] *From* [*Where*] [*Order-by*] [*Union*]
Insert	::=	INSERT *table-name* [(*Insert-list*)] *Values* | INSERT *table-name* (*Query*)
Delete	::=	DELETE *table-name* [*Where*]
Update	::=	UPDATE *table-name* *Update-list* [*Where*]
Select	::=	SELECT [DISTINCT] *Column-list*
Into	::=	INTO *table-name*
From	::=	FROM *Table-list*
Where	::=	WHERE *Condition*
Order-by	::=	ORDER BY *Order-list*
Union	::=	UNION *Query* [*Union*]
Values	::=	VALUES (*Constant-list*)
Condition	::=	*Conjunct* | NOT *Condition* | (*Condition*) | *Condition* AND *Condition*
Conjunct	::=	*Atomic* | *Subquery*
Atomic	::=	*Operand binary-op Operand*
binary-op	::=	< | > | => | <= |! = |! > |! < |*LIKE*
Operand	::=	*Full-Column-name* | *constant*
Subquery	::=	EXISTS (*Query*) | *Full-Column-name* IN (*Query*)
Table-list	::=	*table-name* [*table-name*] [,*Table-list*]
Column-list	::=	*Full-Column-name* [*column-name*] [,*Column-list*]
Insert-list	::=	*column-name* [,*Insert-list*]
Order-list	::=	*Full-Column-name Ordering* [,*Order-list*]
Update-list	::=	SET column-name = constant [,*Update-list*]
Constant-list	::=	*constant* [,*Constant-list*]
Full-Column-name	::=	[*table-name.*] *column-name* | *****
Ordering	::=	ASC | DES
constant	::=	*STRING* | *INTEGER* | **NULL**

Figure 1: Grammar G_{SQL}.

itself. We first present the portion of SQL that will be reflected upon. We next describe and define how a single query is encoded into a table. Lastly, by elaborating upon this design, we show how nested queries, and ultimately programs, are encoded. To motivate the reader's interest, we pose a query that, though reasonable and easily understood, cannot be expressed in pure SQL, in a schema-independent fashion, i.e., if the schema changes, the the SQL query must be rewritten. This prevalent problem is referred to as the *meta-data dependency problem* of SQL. Suppose we have a database, and we ask, "What's known about 'A'?" How can we go about providing an answer? If we have access to an encoding of an SQL query that looks for "A" and upon finding "A", returns all associated data, then we might fashion a collection of queries that utilize the data dictionary, using each row as an argument. The evaluation of this collection of queries would then provide us with our answer. (The complete solution using RSQL is presented in Section 3.)

2.1 The Fragment of SQL

The fragment of SQL (see Fig. 1) we have chosen to reflect is included in the core of SQL, the portion often referred to as the *data manipulation language* [11]. This portion allows the user to manage existing tables by both querying and modifying data. As is well known [17], this fragment is *Codd-complete*, since all relational algebra expressions can be formulated in it.

Fixing on some notation, we reserve the word "operator" to refer to SQL keywords like `SELECT`, `INTO`, and `WHERE`. We will also use "clause" to mean a portion of a *Statement* that has two parts: an SQL keyword and a collection of expressions, generally in the form of a list. A *Select* clause, for instance, is part of a *Query*

```
SELECT R.A F, B, T.C
FROM   R, R S, T
WHERE  D = "string"
       AND 23 < T.E
       AND R.A S.A
```

```
SELECT R.A F, B, T.C
FROM   R, R S, T
WHERE  D = "string"
       AND 23 < T.E
       AND R.A = S.A
       AND EXISTS
              (SELECT *
               FROM   Q
               WHERE  Q.A = R.A
                      AND Z LIKE "%string%")
```

```
SELECT A F
INTO   S
FROM   R
WHERE  B > 44
       AND EXISTS
              (SELECT *
               FROM   Q
               WHERE  R.D = Q.A);

INSERT S (SELECT C
          FROM   T
          WHERE  E != "string")
```

Figure 2: **(Left)** An elementary SQL statement $Q1$. **(Middle)** A nested subquery $Q2$. **(Right)** A program $Q4$.

SQL-op	*t1*	*c1*	*s1*	*n1*	*bin-op*	*t2*	*c2*	*s2*	*n2*
SELECT1	R	A					F		
SELECT2		B							
SELECT3	T	C							
FROM	R								
FROM	R					S			
FROM	T								
WHERE		D			=			string	
WHERE				23	<	T	E		
WHERE	R	A			=	S	A		

Figure 3: The encoding of $Q1$ into a table **P1**.

statement and contains the operator `SELECT` and a list of column names. Lastly, we let "query" mean either a *Query* or *Select-Into* statement, unless otherwise noted.

2.2 Encoding of SQL statements

Our discussion here will precede informally, giving the intuition behind encoding SQL statements into a distinguished class of tables called *program tables* (*p-tables*). Initially we will describe how *elementary* SQL *Query* statements are encoded into tables. With this in hand we will describe how to encode more general queries into *p-tables*. Our discussion will conclude with a description of the encoding of a program. Formal aspects of encoding and evaluation will be discussed in Section 3.

In *elementary Query* statements, the condition in the *Where* clause is, when it exists, the conjunction of *Atomic* expressions. With this in mind, we begin with Example 2.1 where we encode an elementary query.

Example 2.1 (Encoding an elementary query) The query $Q1$ in Fig. 2 (Left) contains information on how it can itself be encoded. For example, there are three clauses, i.e., `SELECT`, `FROM`, and `WHERE`, the *Select* clause has three table expressions, where `R.A F` is the first of these expressions, `B` is the second, and so forth. We use this information to encode $Q1$ into a table **P1** as shown in Fig. 3, incorporating all the information sufficient to rebuild $Q1$. ■

Each *Select*, *From*, and *Where* clause corresponds to a particular number of rows, the number of rows depending upon the number of *Column-list*, *Table-list*, and *Atomic* expressions they contain, respectively. In this query, each clause contains three expressions, and therefore, there are nine rows. An operator is placed in the *SQL-op* column identifying the clause to which the expression belongs. Those entries that do not contain values we assume to have some fixed, blank datum.

The *SQL-op* column contains a keyword signifying the kind of clause. Columns *t1* and *t2* contain table names, and *c1* and *c2* contain column names. If, in a particular row, *t1* and *c1* or *t2* and *c2* are non-blank, then we infer the table-column combination is a dotted pair. String constants are held in the *s1* and *s2* columns, and numeric constants are held in the *n1* and *n2* columns. The binary operator in an *Atomic* expression is placed in the *bin-op* column.

We are also relying upon the *declarative* nature of SQL as we encode a statement. One of the prime advantages of working with a declarative language is the freedom we are given to rearrange expressions without affecting the expression's value — in this case, we can arrange *any* expressions that do not affect the result of the query. To underscore this fact we might rearrange $Q1$'s *From* clause as `FROM R S,T,R` or `FROM T,R S,R`, neither changing the result of the query. This applies to the order of the *Atomic* formulae as well. Since the order of expressions in the *Column-list* is important, however, an integer is postfixed to `SELECT` indicating the position of its expression, i.e., whether it is first, second, and so on. `SELECT3 T C` indicates, for example, that `T.C` is the third member of the *Column-list.* Ordering is immaterial for both the *From* and *Where* clauses, as we have pointed out, and therefore, neither `FROM` nor `WHERE` are numbered.

Observe that our mapping from query to table is, indeed, an encoding. While the ordering of **P1**'s rows themselves correspond to $Q1$'s ordering of the expressions in their respective clauses, this need not be the case. An SQL statement *equivalent* to $Q1$ can be "reassembled" from any ordering of rows, as long as we have reassembled the *Select* clause in its original order and gathered the *From* and *Where* expressions correctly. Interestingly, now that we have a table, we are free to operate upon this table using queries again (see Remark 2.2).

Remark 2.2 (Manipulating encodings of queries) Given **P1** from the previous example, we can manipulate **P1** via a query to create an encoding of a new query. For example, selecting from **P1** where *bin-op* does not contain = and *SQL-op* is not "SELECT3" results in an encoding of `SELECT R.A F,B FROM R,R S,T WHERE 23 < T.E` ■

Although we have successfully encoded elementary queries, a much richer class of SQL statements remain. Further, we have provided only the *necessary* schema. With Example 2.3, we present a schema that is *sufficient* for encoding all statements in G_{SQL}, and in particular, any query, by encoding a nested SQL query.

Example 2.3 (Encoding a nested SQL query) Query $Q2$ (see Fig. 2) (Middle) is our original query $Q1$ from Ex. 2.1 (the outer query) and an additional existential predicate and subquery (the inner query). (Note that the number of conjuncts in $Q3$'s *Where* clause has increased from three to four.) We can immediately encode the subquery itself as **P2** shown in Fig. 4. What is left for us to do is to combine the two tables

SQL-op	*t1*	*c1*	*s1*	*n1*	*bin-op*	*t2*	*c2*	*s2*	*n2*
`SELECT1`		`*`							
`FROM`	`Q`								
`WHERE`	`Q`	`A`			=	`R`	`A`		
`WHERE`		`Z`			LIKE			`%string%`	

Figure 4: The encoding of the subquery of $Q2$, the inner query, into **P2**.

P1 and **P2** into a single *p-table* while capturing the fact that **P2** is the argument to an *EXISTS* predicate, the whole of which comprises the fourth conjunct of $Q2$'s *Where* clause. We accomplish this by adding two more columns *id* and *expr-id*, and a new SQL keyword `WHERE_EXISTS`. The column *id* contains an *identifier*, an integer, signaling that rows sharing this value belong to a single expression. The column *expr-id* contains an *expression identifier*, an integer, indicating the *id* of one or more rows that together form an argument to, or an expression of, the *SQL-op* operator. We now can encode $Q2$ into **P3** shown in Fig. 5. ■

id	*SQL-op*	*t1*	*c1*	*s1*	*n1*	*bin-op*	*t2*	*c2*	*s2*	*n2*	*expr-id*
1	`SELECT1`	`R`	`A`					`F`			0
1	`SELECT2`		`B`								0
1	`SELECT3`	`T`	`C`								0
1	`FROM`	`R`									0
1	`FROM`	`R`					`S`				0
1	`FROM`	`T`									0
1	`WHERE`		`D`			=			`string`		0
1	`WHERE`				23	<	`T`	`E`			0
1	`WHERE`	`R`	`A`			=	`S`	`A`			0
1	`WHERE_EXISTS`										2
2	`SELECT1`		`*`								0
2	`FROM`	`Q`									0
2	`WHERE`	`Q`	`A`			=	`R`	`A`			0
2	`WHERE`		`Z`			LIKE			`%string%`		0

Figure 5: The encoding of $Q2$ into the *p-table* $P3$.

P3 has two new columns, *id* and *expr-id*. Those rows forming $Q2$'s outer query have $id = 1$, and those of the inner query have $id = 2$. The means of assignment is discussed fully in Section 3, but it is sufficient to say here that the outermost *id* is assigned the smallest value in the table, and *id* values strictly increase as we move inward to subexpressions. We noted previously that the number of rows in an encoding depends directly upon the number of expressions each clause contains. $Q2$ contains four conjuncts, and there are, indeed, four rows where $id = 1$ and *SQL-op* is a string prefixed by "WHERE". The new row contains `WHERE_EXISTS` as its operator. This operator codes for a conjunct that, as an expression, applies the *EXISTS* predicate to a query. The query in this case has $expr\text{-}id = 2$, matching the *id*s that when taken together form $Q2$'s inner query. If the row does not contain a subexpression, then $expr\text{-}id = 0$.

As before, the ordering of **P3**'s rows is immaterial, since sufficient information resides in the table to reconstruct an SQL statement equivalent to $Q3$. Now that we have statements well at hand, we can encode programs.

Recall that a program is a sequence of statements. Encoding programs, illustrated in Example 2.4, is the repeated encoding of statements with the following constraint: encoding a statement S_i requires that S_i's minimal *id* value is strictly less than any following statement S_{i+1}'s minimal *id* value.

Example 2.4 (Encoding an SQL program) Fig. 2 (Right) contains a program $Q4$ comprised of two SQL statements. $Q4$ is encoded into **P4** shown in Fig. 6. Note that the first statement's *id* = 1 and the second's *id* = 3. We have not used the *Insert* statement until now, but encoding is based upon the statement's grammar. (Note we have opted to use the shorter `INSERT` instead of `INSERT INTO`.) Referring to G_{SQL}, an *Insert* statement requires a table name and, in this case, a query. We place `INSERT` into *SQL-op*, a table name into *t1*, and an *id* into *expr-id*, where the query's *id* matches the *expr-id*. ■

id	*SQL-op*	*t1*	*c1*	*s1*	*n1*	*bin-op*	*t2*	*c2*	*s2*	*n2*	*expr-id*
1	SELECT1		A					F			0
1	INTO	S									0
1	FROM	R									0
1	WHERE		B			$>$				44	0
1	WHERE_EXISTS										2
2	SELECT1		*								0
2	FROM	Q									0
2	WHERE	R	D			$=$	Q	A			0
3	INSERT	S									4
4	SELECT1		C								0
4	FROM	T									0
4	WHERE		E			$!=$			string		0

Figure 6: The encoding of $Q2$ into the *p-table* $P4$.

We next discuss two, new operators that are added to SQL: *reify* serves to encode programs into *p-tables* and *eval*, reify's "complement", serves to decode and evaluate *p-tables.*

3 Reification and Evaluation

Adding *reify* and *eval* involves modifying G_{SQL} at both the statement and the clause level. We will refer to this new grammar as G_{RSQL}. Instead of reproducing G_{RSQL} in its entirety—most of the grammar remains unchanged—we will present the extensions by themselves.

Reflective-program	::=	*Reflective-statement* ; [*Reflective-program*]
Reflective-statement	::=	*Statement* $\in G_{\mathrm{SQL}}$ | *reify* | *eval*
Reify	::=	REIFY (*R-expr*) INTO *table-name* [*Starting-value*]
Eval	::=	EVAL *P-table* [EVAL-INTO *table-name*]
R-expr	::=	*Program* | *Select* | *Into* | *From* | *Where* | *Union* | *Order-by*
Starting-value	::=	ID = *Value-id* | ID > *Value-id*
Value-id	::=	*NATURAL* | (*Aggregate-Query*)
P-table	::=	*table-name* | (*Select-Into*)
R-From	::=	FROM *R-Table-list*
R-Table-list	::=	*R-Table-expr* [, *R-Table-list*]
R-Table-expr	::=	(*Reify*) | (EVAL *P-table* [*Eval-into table-name*]) | *table-name* [*table-name*]
Eval-into	::=	EVAL-INTO | TEMP-EVAL-INTO

A reflective program is a sequence of reflective statements S, where S is either a *Statement* $\in G_{\mathrm{SQL}}$, *reify*, or *eval.* For any non-*reify* or non-*eval* reflective statement, the *From* clause has been extended (*R-From*) to allow reification and evaluation in its list of table names, over which we have placed an explicit order evaluation. We now take a closer look at *reify* and *eval* as statements.

Reification can take place only over expressions from G_{SQL}, i.e., *R-expr* $\in G_{\text{SQL}}$. The table into which the program is reified cannot exist prior to the statement's execution. Including the *Starting-value* clause allows reification to begin at either a particular *id* or its successor, its value being either an explicit natural number or the result of an aggregate query. We show a typical *reify* statement in Example 3.1.

Example 3.1 (Reifying statements) Ex. 2.4 showed the reification of two statements. We might have reified the statements separately. After initially reifying the first statement into **P4**, we continue as shown in Fig. 7 (Left). First, the *Insert* statement is reified into **Temp**. Next, the contents of **Temp** are copied into **P4**. ■

Evaluation can take place only over expressions that are equivalent to properly reified expressions. These are obtained either directly through reification or indirectly by properly manipulating tables. The *eval* operator can be supplied with either a table name **T** that holds a program or a query whose result is a *p-table*. Suppose **T** is a reified program of $S_1; \ldots ; S_k$ of k statements. Including an EVAL-INTO *table-name* clause allows the results of statement S_1, if it is a query, to persist, saving the query in *table-name*. It is important to note that the *Into* clause is different from EVAL-INTO. The *Into* clause makes some portion of the *p-table* persistent, functioning as it normally would, whereas, EVAL-INTO makes the first query of the *p-table* persistent. To illustrate this point, we provide Example 3.2.

Example 3.2 (Making results persistent) Suppose we have a table **Emp**(**Name**,**Eno**,**Sal**) of employees containing names, employee numbers, and salaries. Say we reify into **S** a query that finds certain employees. Sometime after that we want to execute only a portion of **S**, also wanting to save this new, dynamically created query as **R**. Furthermore, we want to save the results of running **R**. This can be easily done as shown in Fig. 7 (Middle). **S** contains the original query, **R** contains a modified **S** in which the condition on the name is removed, and **T** contains the results of executing **R**. ■

Now that we have been introduced to *reify* and *eval*, we can make it clear how these statements can play the dual role of expression in a clause. In an *R-From* clause, the order of evaluation depends directly upon the expression's location, i.e., the first expression will be evaluated first, and so on. A *reify* in an *R-From* clause will create a single, persistent table—the table name following the *reify*'s INTO—that contains a reified expression and leaves no trace of its execution in the *R-From* clause. An *eval* operates in one of three, related ways. EVAL *P-table* evaluates its argument, but leaves no trace of execution in the *R-From* clause. EVAL with EVAL-INTO evaluates its argument and functions as a single, persistent table—the table name following EVAL-INTO—the contents of which contain the result of the *p-table*'s first query. Lastly, EVAL with TEMP-EVAL-INTO evaluates its argument, but only functions as an aliased table—the table name following TEMP-EVAL-INTO—the contents of which contain the result of the *p-table*'s first query. The next example shows *eval* as a part of a clause.

```
REIFY (INSERT S
         (SELECT C
          FROM   T
          WHERE  E != "string"))
INTO Temp
ID > (SELECT MAX(id) FROM P4);

INSERT P4 (SELECT * FROM Temp);
```

```
REIFY (SELECT Name
       FROM   Emp
       WHERE  Name LIKE "A%"
              AND Sal > 10)
INTO S;

EVAL  (SELECT *
       INTO   R
       FROM   S
       WHERE  bin-op != "LIKE")
EVAL-INTO T;
```

```
SELECT P.Name, Job
FROM   (EVAL R TEMP-EVAL-INTO P), WA
WHERE  P.Name = WA.Name
       AND Sal < 20;
```

Figure 7: **(Left)** Reifying statements separately. **(Middle)** Saving the results of *eval*. **(Right)** Aliased *eval* result.

Example 3.3 (EVAL in the FROM clause) Continuing from Ex. 3.2, suppose we have another table **WA**(**Name**,**Job**) containing employee names and work assignments. We want to find the jobs of those employees selected by evaluating **R**, additionally narrowing the search by including **Sal** < 20; however, we do not want the result to persist. The RSQL statement is shown in Figure 7 (Right). The *R-From* clause here has two expressions: `(EVAL R TEMP-EVAL-INTO P)` is the first and `WA` is the second. **P** is aliased to the result of evaluating **R** and exists only during the run of program—if we had used EVAL-INTO, the table would have become persistent. ■

So far, we have not witnessed the extent of *eval*'s expressiveness. Before we do, we must briefly discuss the *eval*'s functional details in order to better appreciate its operation.

Ordering is essential to the mechanics of *eval*. We assume a total ordering over the set of column names C and over the universe U, where $U = \cup_{i=1}^{\infty} C_i$. Using these assumptions, we can order any two rows t_i, t_j from some table **T**. At outset of Section 2.2, we stated that *p-tables* are a class of tables. We now make this clear. Any table that *includes* the schema of a *p-table*, i.e., $\{id, SQL\text{-}op, t1, c1, s1, n1, bin\text{-}op, t2, c2, s2, n2, expr\text{-}id\}$ is itself a *p-table*. The extra column names $C_1, C_2, \ldots, C_k$ we consider to be an *extension* of the *id* value, and herein lies why ordering is so important: we can first order on the column names themselves, then order over the rows, thus, ordering the expressions much as we did with the *id* alone, viz.,

$$\Leftarrow 1^{st}\ Ordering$$

$$2^{nd}\ Ordering \Downarrow \qquad \underbrace{\underbrace{\overbrace{C_1\ C_2\ \ldots\ C_k}^{\text{initial segment}}\ id}_{extended\ id}\ SQL\text{-}op\ \ldots\ id\text{-}expr}_{p-table}$$

We now complete this section with examples that utilize this ordering assumption. The first demonstrates an iterative construct.

Example 3.4 (Finding the descendents of "A") Suppose we have a table **R**(**Parent**,**Child**) containing parents and their children, and we want to find *all* the descendents of "A". We first discuss our strategy, then present the actual code. Suppose we have some table, say, **Desc**, that contains only "A", and a statement that inserts into **Desc** all of the immediate descendents of **Desc**. If we can somehow *repeat* this statement as many times as there are rows in **R**, then we will have accumulated all of the descendents of "A". We first create a table **Desc**(**Child**), placing into it, the sole progenitor "A". Figure 8 (Left) shows

```
REIFY (INSERT Desc
         (SELECT Child
          FROM   R
          WHERE  EXISTS
                   (SELECT *
                    FROM   Desc
                    WHERE  R.Parent = Desc.Child)))
INTO Q;

SELECT DISTINCT *
FROM   (EVAL (SELECT * FROM R, Q)), Desc
WHERE  Child != "A";
```

```
SELECT D.Tn, D.Cn1, D1.Cn1 Cn2
INTO   D2
FROM   D, D D1
WHERE  D.Tn = D1.Tn AND D1.Ct = "string";

REIFY  (SELECT Tn, Cn1, [Cn1] Val
        FROM   D2, [Tn]
        WHERE  "A" = [Cn2]
               AND Tn = "[Tn]"
               AND Cn1 = "[Cn1]"
               AND Cn2 = "[Cn2]")
INTO Q;
```

Figure 8: **(Left)** An RSQL program that finds the descendents of "A". **(Right)** The initial RSQL statements to find, "What's known about 'A'?"

```
...
SELECT Tn, Cn1, Sal Val
FROM   D2, Emp
WHERE  "A" = Name AND Tn = "Emp" AND Cn1 = "Sal" AND Cn2 = "Name"
...
SELECT Tn, Cn1, Child Val
FROM   D2, R
WHERE  "A" = Parent AND Tn = "R" AND Cn1 = "Child" AND Cn2 = "Parent"
...
```

```
 Tn          Cn1          Val
----------------------------
 ...
 "Emp"       "Sal"        10
 ...
 "R"         "Child"      "B"
 ...
```

Figure 9: **(Left)** A portion of the SQL program that the *p-table* in STEP 3 of Ex. 3.5 encodes. **(Right)** Some of the results of evaluating STEP 3.

the remaining two steps. The first RSQL statement reifies into **Q** the means of adding into **Desc** all of the immediate descendents of **Desc**. The second statement evaluates a program containing $\|\mathbf{R}\|$ copies of **Q**, selecting after this evaluation, all the descendents accumulated in **Desc**—in other words, "looping" $\|\mathbf{R}\|$ times with body **Q**. ■

The *p-table* created in Fig. 8 has, as its initial segment, tables **Parent** and **Child**. After ordering these two tables lexicographically, each row of **R**, together with an *id* value, becomes an extended *id*. This new, *extended id* gives us sufficient information to reconstruct the program. The next example illustrates how schema-independent queries can be written in RSQL.

Example 3.5 (What's known about "A") Continuing from Example 3.4, Suppose we want to find out everything there is to know about "A"; that is, for every row t in every table $\mathbf{T}$ that contains "A", we want a triple $\langle \mathbf{T},\mathbf{C},v \rangle$, where $t.\mathbf{C} = v$. Using RSQL we can exploit the *data dictionary*, a table that holds information about the database's tables and columns. For this example, we assume a data dictionary $\mathbf{D}(\mathbf{Tn},\mathbf{Cn},\mathbf{Ct})$, where **Tn** is a table name, **Cn** is a column of **Tn**, and **Ct** is the column type, i.e., string, natural, etc.,. We will outline our strategy then present the code. The idea is akin to writing a parameterized query. The query will expect three values: a table name and two column names. The "filling-in" is accomplished in much the same manner as the previous example where we made an appropriate number of copies of a statement. In this case, however, by joining we not only make copies, but also use the row *itself* as an argument to the query. We now present the procedure:

1. We first query the data dictionary, shown in the first RSQL statement in Figure 8 (Right). **D2** now contains the triples $\langle\ \mathbf{T},\mathbf{C_i},\mathbf{C_j}\ \rangle$. Each row will form an argument to a query that requires a table name

and two column names.

2. We reify a statement—the second RSQL statement in Figure 8 (Right)—that will form a kind of template: each [Tn], [Cn1], and [Cn2] is to be replaced with its respective value from every row in **D2**.
3. Lastly, make $\|\mathbf{D2}\|$ copies of the statement, simultaneously replacing each [Tn], [Cn1], and [Cn2] with its respective value $t.\mathbf{T}$, $t.\mathbf{C_i}$, and $t.\mathbf{C_j}$, for every $t \in \mathbf{D2}$ (see Fig. 9):

```
EVAL  (SELECT Tn,Cn1,Cn2,id,SQL-op,t1,Cn1 c1,s1,n1,bin-op,t2,c2,s2,n2,expr-id
       FROM   D2,Q
       WHERE  c1 = [Cn1]
     UNION
       SELECT Tn,Cn1,Cn2,id,SQL-op,Tn t1,c1,s1,n1,bin-op,t2,Cn2 c2,s2,n2,expr-id
       FROM   D2,Q
       WHERE  c2 = [Cn2]
     UNION
       SELECT Tn,Cn1,Cn2,id,SQL-op,Tn t1,c1,s1,n1,bin-op,t2,c2,s2,n2,expr-id
       FROM   D2,Q
       WHERE  t1 = [Tn]
     UNION
       SELECT Tn,Cn1,Cn2,id,SQL-op,t1,c1,s1,n1,bin-op,t2,c2,Tn s2,n2,expr-id
       FROM   D2,Q
       WHERE  s2 = "[Tn]"
     UNION
       SELECT Tn,Cn1,Cn2,id,SQL-op,t1,c1,s1,n1,bin-op,t2,c2,Cn2 s2,n2,expr-id
       FROM   D2,Q
       WHERE  s2 = "[Cn2]"
    UNION
       SELECT Tn,Cn1,Cn2,id,SQL-op,t1,c1,s1,n1,bin-op,t2,c2,Cn1 s2,n2,expr-id
       FROM   D2,Q
       WHERE  s2 = "[Cn1]"
    UNION
       SELECT Tn,Cn1,Cn2,id,SQL-op,t1,c1,s1,n1,bin-op,t2,c2,s2,n2,expr-id
       FROM   D2,Q
       WHERE  c1 != [Cn1] AND c2 != [cn2] AND t1 != [Tn] AND s2 != "[Tn]" AND s2 != "[Cn1]" AND s2 != "[Cn2]");
```

■

Example 3.6 (What is known about *anyone*) Continuing from Ex. 3, suppose we had reified the statements in STEP 1 and STEP 3 as **V** and **S**, respectively. We now can search for *anyone* X with a modest amount of code (we can use the same technique in Ex. 3.4):

```
UPDATE Q SET s1 = X
         WHERE s1 = "A";
EVAL V;
EVAL S;
```

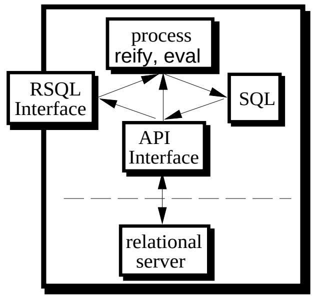
■

4 Implementation

The implementation of RSQL is comprised of two parts: an RSQL interface and a relational database server (see Figure 10 (Left)). The arrows provide the flow of events.

The interface—an interactive RSQL environment—is written in the Scheme programming language [8], since it is particularly well-suited for rapid prototyping. In keeping with the spirit of the language, we slightly adapted the syntax of RSQL by substituting *lists* for comma delimited expressions and by using prefix notation in place of infix. For example, `FROM C1,C2` is written as `FROM (C1 C2)` and `(C1 > 23)` is written as `(> C1 23)`.

After a statement is entered, the interface looks for *reify* and *eval* expressions as it transliterates the statement. If there are none, the transliterated statement, now an SQL string, is submitted to the server



```
> (REIFY (INSERT Desc (SELECT (Child)
                       FROM   (R)
                       WHERE  (EXISTS (SELECT (*)
                                       FROM   (Desc)
                                       WHERE  (= (R.Parent) (Desc.Child)))))))

> (SELECT DISTINCT (*)
   FROM   ((EVAL (SELECT (*)
                  FROM   (R Q)))
           Desc)
   WHERE (!= Child "A"))
```

Figure 10: (Left) System architecture. (Right) Sample session from Example 3.4.

via the API interface. If there are *reify* or *eval* expressions, additional data and schema data are read from and written to the server before transliteration of the original statement is completed and submitted to the server. It must be emphasized that the interface does no *real* work: all persistent and intermediate tables are stored in the server, and, furthermore, all table manipulation is carried out by the server. The server uses a native extension of SQL that allows multiple statements to be submitted per transaction, and all exchanges between the interface and server is conducted in this native code. Figure 10 (Left) shows the code from Example 3.4 as it looks in our implementation.

5 Conclusions and Future Work

In this paper we show how reflection can be easily admitted into the relational model, and in particular, SQL. A highlight of our work is an encoding system that is easily understood, managed and handled, retaining key features of the encoded language, e.g., declarativeness. Our aim in establishing RSQL is to enhance SQL sufficiently enough to widen the scope of potential problems programmers can solve without concomitantly giving up properties of the relational model that make it so widely studied and used in the first place. We have added reflection to SQL using this encoding scheme and two, readily understood operations: *reify* that encodes programs into data—in our case into ordinary SQL tables and *eval* that evaluates the statements held by the tables. Because of the closure property, we have at our disposal the full use of the language itself, something we presumably well understand. In addition, since a program can examine its own contents, i.e., "introspect", the program itself can modify its own behavior. We witnessed some examples of how powerful this reflection can be, viz., the simulation of looping and schema-independent querying. Finally, we have implemented a interactive Reflective SQL system comprised of an RSQL interface and a relational database server.

A natural direction for future work is to incorporate reify and eval in the encoding itself thereby making the language more general. Another aspect that can benefit from reflective mechanisms is transaction processing where we envision a rule as a *dynamic-eval* that gets triggered upon updates. Additional future work lies in implementing other applications of reflection in databases [18, 6].

References

[1] Aho, A., and Ullman, J. Universality of data retrieval languages. In *Sixth Annual ACM Symposium on Principles of Programming Languages, Williamsburg* (1979), pp. 110–117.

[2] Chandra, A., and Harel, D. Structure and complexity of relational queries. *J. Comput. Syst. Sci. 25*, 1 (1982), 99–128.

[3] Codd, E. A relational model for large shared data banks. *Commun. ACM 6*, 13 (June 1970), 377–387.

[4] Codd, E. Relational completeness of database sublanguages. In *Database Systems*, R. Rustin, Ed. Prentice-Hall, Englewood Cliffs, N.J., 1972.

[5] Immerman, N. Relational queries computable in polynomial time. *Information and Control 68* (1986), 86–104.

[6] Jain, M. Unpublished manuscript. Available from the author.

[7] Jain, M., Mendhekar, A., and Van Gucht, D. The uniform data model for relational data and meta-data query processing. In *Proceedings of the Seventh International Conference on Management of Data* (December 1995), pp. 146–165.

[8] Kent Dybvig, R. *The SCHEME Programming Language.* Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[9] Krishnamurthy, R., Litwin, W., and Kent, W. Language features for interoperability of databases with schematic discrepancies. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data* (May 1991), pp. 40–49.

[10] Lakshmanan, L., Sadri, F., and Subramanian, I. On the logical foundations of schema integration and evolution in heterogeneous database systems. In *Proceedings of the 3rd International Conference on Deductive and Object-Oriented Databases* (December 1993), pp. 81–100.

[11] Melton, J., and R. Simon, A. *SQL: A Complete Guide.* Morgan Kaufmann, San Francisco, 1993.

[12] Smith, B. C. Reflection and semantics in a procedural language. Tech. Rep. MIT-LCS-TR-272, MIT, Cambridge, 1982.

[13] Smith, B. C. Reflection and semantics in lisp. In *Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City* (Jan. 1984), pp. 23–35.

[14] Stemple, D., et al. Type-safe linguistic reflection: a generator technology. Research Report CS/92/6, University of St Andrews, Department of Mathematical and Computational Sciences, 1992.

[15] Stonebraker, M., Anderson, E., Hanson, E., and Rubenstein, B. QUEL as a data type. In *Proceedings of ACM-SIGMOD 1984 International Conference on Management of Data, Boston* (1984), B. Yormark, Ed., vol. 14:2 of *SIGMOD Record*, ACM Press, pp. 208–214.

[16] Stonebraker, M., Anton, J., and Hanson, E. Extending a database system with procedures. *ACM Trans. Database Syst. 12*, 3 (Sept. 1987), 350–376.

[17] Ullman, J. D. *Principles of Database Systems*, 2 ed. Computer Science Press, Rockville, MD, 1982.

[18] Van den Bussche, J., Van Gucht, D., and Vossen, G. Reflective programming in the relational algebra. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Washington, D.C.* (May 1993), pp. 17–25.

[19] Vardi, M. Complexity and relational query languages. In *Proceedings of the Fourteenth Annual Symposium on Theory of Computing, San Francisco* (1982), pp. 137–146.