MD-SQL: A Language for Meta-Data Queries over Relational Databases

C.M. Rood
Indiana University
Computer Science Dept.
crood@cs.indiana.edu

D. Van Gucht
Indiana University
Computer Science Dept.
vgucht@cs.indiana.edu

F.I. Wyss
Interactive Intelligence, Inc.
Indianapolis, IN
felixw@inter-intelli.com

July 16, 1999

Abstract

Future users of large, interconnected databases and data warehouses will increasingly require *schematic transparency* of data manipulation systems, in that (i) data from heterogeneous sources must be compared and interrelated and (ii) data must be queried and extracted by distant users having minimal knowledge of its logical structure. A query language that abstracts over meta-data as well as ordinary data is needed. Previous work in this area has resulted in HILOG [1], SchemaLog [7] and SchemaSQL [8]. Although SchemaSQL improves on its predecessors, it remains somewhat informal and relies on a specialized transformation into a fragment of the tabular algebra [10] to give it a viable operational semantics. In contrast, we provide a complete EBNF for *Meta-Data SQL* (MD-SQL) as a straightforward extension of a relationally complete subset of standard SQL. Like SchemaSQL, MD-SQL allows queries involving meta-data and ordinary data in a multi-database context over potentially disparate platforms. Schematic elements and data are freely interchangeable, and queries are allowed whose output type cannot be known at compile time. Unlike SchemaSQL however, each MD-SQL query translates into a series of simple, atomic operations, each of which is inherently relational. We formalize this translation by presenting a complete *meta-algebra* which is shown to be equivalent to MD-SQL. Furthermore, we provide some complexity results, in particular that MD-SQL and the meta-algebra yield characterizations of PSPACE. We also give results concerning when the output type of an MD-SQL query can be deduced at compile time. Finally we briefly discuss an implementation of MD-SQL over an ordinary, relational system that uses the DynamicSQL/CLI standard. Since MD-SQL is relational in nature, our implementation can benefit directly from existing query optimization techniques.

1 Introduction

Thirty years ago, the relational model was introduced as an abstraction away from the physical organization of stored data, enabling differing platforms to look similar at the logical level. Similarly, one goal of a meta-data query framework is to abstract away from the exact schematic organization of the data, allowing differing schemas to appear interchangeable at the logical (or view) level. To this end, we present the language *Meta Data SQL*, or MD-SQL, which is equivalent to a powerful, relation based *meta-algebra*. These formalisms yield a uniform framework for defining precisely what a meta-data query over a relational system is and how it can be implemented. Applications of MD-SQL are many and varied, including:

Common query processing. As discussed at length in [10], the ability to write queries that extract meta-data (or refer to it interchangeably with ordinary data) facilitates *schema independent querying*, useful (for example) in any context where the end user cannot be expected to be familiar with the exact structure of the data [12].

Heterogeneous data sources. Part of connecting different data sources involves compensating for the fact that similar information may be stored in disparate ways in different databases. MD-SQL is defined in a multi-database context in a manner commensurate with the demands of inter-operability and data restructuring.

Data mining/warehousing. The extraction of meta-data, or restructuring of legacy data is a natural application for any meta-data query language; MD-SQL is no exception.

Although it has diverged significantly from SchemaSQL, MD-SQL has its roots in that language. The applications of SchemaSQL indicated in [8] and [10] are natural for MD-SQL.

In the remainder of this section, we present our data model and examples of the types of meta-queries MD-SQL supports. Much of this section covers ground already discussed in [8] and [10], however subsequent sections move into new territory. At the end of this section we indicate our main contributions and give a brief overview of the rest of the paper.

Each MD-SQL query takes place in the context of a unique *federation* which consists of several (ordinary relational) databases. Whereas an SQL query maps a single database instance (set of relation instances) to a single output relation instance, an MD-SQL query maps a federation instance (*set* of database instances) to an output database (*set* of relation instances). Note that in general an MD-SQL query will be *untyped*, in that the output schema cannot be determined at compile time (§5.2). For convenience (compatibility with ordinary SQL), we allow an MD-SQL query to output a single relation.

As examples of the types of queries MD-SQL supports, refer to the university federation in figure 1 where several universities keep track of the average salaries of various categories of employees in schematically differing databases.[1] We can phrase queries regarding the information in one or more of the databases, such as "Which employees of departments of university `A` are paid the same average salary as their counterparts in university `B`?" or

[1]This federation is a slightly changed version of the one appearing in [8].

univ-A:
`salInfo`:

category	dept	avgSal
prof	cs	60,000
assoc_prof	cs	50,000
prof	math	65,000
assoc_prof	math	45,000

univ-B:
`salInfo`:

category	cs	math
prof	60,000	60,000
assoc_prof	45,000	50,000

univ-D:
`salInfo`:

dept	prof	assoc_prof
cs	65,000	55,000
math	60,000	50,000

univ-C:
`cs`:

category	avgSal
prof	62,000
assoc_prof	55,500

`math`:

category	avgSal
prof	65,000
assoc_prof	52,000

Figure 1: University federation.

"Return the relations from the university `C` database for those departments that pay their professors an average salary of over $58,500." The latter query is *untyped* in that the number of output relations is unknown at compile time. Another untyped query is: "restructure the information in the university `C` database into the format of the university `B` database."[2]

The main contributions of this paper are as follows. First, we give a formal characterization of meta-data queries over relational databases, both in terms of a declarative query language (MD-SQL, §2) and an algebra (§3). We prove the equivalence of the two formalisms (§4), yielding a method for algebratizing MD-SQL queries. We then show that MD-SQL and the meta-algebra characterize PSPACE, and that in general one cannot deduce the output schema of an MD-SQL query at compile time (§5). Finally, we discuss an implementation of MD-SQL in a relational context (§6). One requirement for our framework was that it stay as close as possible to the relational model. MD-SQL extends this model, but can be seamlessly implemented over existing relational platforms using the DynamicSQL/CLI standard.

2 MD-SQL

Like SchemaSQL, MD-SQL is a second-order language with a first-order semantics. We assume a single underlying domain, **dom**. Each element of **dom** is a valid *meta-name*, i.e. the name of a column, relation or database.[3] Relations are primary; each relation must be named. A relation name is a *path name*, consisting of a database name and a relation name, written `dbName::relName` where "`dbName`" and "`relName`" are elements of **dom**.

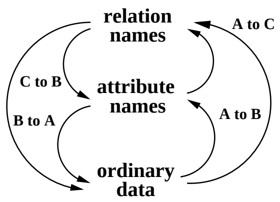

Figure 2: Meta-data translations.

Figure 2 shows the translations MD-SQL supports among meta-data and ordinary data. Data can float up and down among the three tiers transparently. Certain arcs in figure 2 are labeled; the MD-SQL for these example translations is given below.

[2]These are ordinary MD-SQL queries (see §2 for examples), whereas SchemaSQL uses a "`create view`" statement to deal with untyped and restructuring queries.

[3]In MD-SQL, data can be of numeric type. If an attempt is made to promote a number to a relation name, we have two options: disallow the promotion, or cast the integer into a valid SQL name, such as "x_123". The current implementation does the former, even though the latter is more uniform (and sometimes desirable).

$\langle \mathit{query} \rangle$	::=	**SELECT** ($\langle \mathit{relation\_decls} \rangle$ | $\langle \mathit{column\_decls} \rangle$)
		[**FROM** $\langle \mathit{variable\_decl} \rangle$ {**,** $\langle \mathit{variable\_decl} \rangle$}*]
		[**WHERE** $\langle \mathit{condition} \rangle$ {**AND** $\langle \mathit{condition} \rangle$}*]
	|	**(** $\langle \mathit{query} \rangle$ **)** {**UNION** **(** $\langle \mathit{query} \rangle$ **)**}$^+$
$\langle \mathit{relation\_decls} \rangle$	::=	$\langle \mathit{relation\_decl} \rangle$ {**,** $\langle \mathit{relation\_decl} \rangle$}*
$\langle \mathit{relation\_decl} \rangle$	::=	**(** $\langle \mathit{column\_decls} \rangle$ **)** **AS** $\langle \mathit{domain\_expr} \rangle$
$\langle \mathit{column\_decls} \rangle$	::=	$\langle \mathit{column\_decl} \rangle$ {**,** $\langle \mathit{column\_decl} \rangle$}*
$\langle \mathit{column\_decl} \rangle$	::=	$\langle \mathit{variablename} \rangle$ **. ***
	|	$\langle \mathit{domain\_expr} \rangle$ [**AS** $\langle \mathit{domain\_expr} \rangle$]
	|	**ALL** $\langle \mathit{domain\_expr} \rangle$ [**AS** $\langle \mathit{domain\_expr} \rangle$]
$\langle \mathit{variable\_decl} \rangle$	::=	**->** $\langle \mathit{variablename} \rangle$
	|	$\langle \mathit{meta\_atom} \rangle$ **->** $\langle \mathit{variablename} \rangle$
	|	[$\langle \mathit{meta\_atom} \rangle$] **::** $\langle \mathit{meta\_atom} \rangle$ [**->**] $\langle \mathit{variablename} \rangle$
	|	[**ALL**] **(** $\langle \mathit{query} \rangle$ **)** [**->**] $\langle \mathit{variablename} \rangle$
$\langle \mathit{condition} \rangle$	::=	**(** $\langle \mathit{condition} \rangle$ **)**
	|	$\langle \mathit{cond\_expr} \rangle$ $\langle \mathit{cond\_operator} \rangle$ $\langle \mathit{cond\_expr} \rangle$
	|	[**NOT**] **EXISTS** **(** $\langle \mathit{query} \rangle$ **)**
$\langle \mathit{cond\_operator} \rangle$	::=	**=** | **!=** | **<>** | **<=** | **<** | **>** | **>=**
$\langle \mathit{cond\_expr} \rangle$	::=	$\langle \mathit{domain\_expr} \rangle$ | $\langle \mathit{integer} \rangle$ | $\langle \mathit{real} \rangle$
$\langle \mathit{domain\_expr} \rangle$	::=	$\langle \mathit{string} \rangle$ | $\langle \mathit{variablename} \rangle$**.**$\langle \mathit{meta\_atom} \rangle$ | $\langle \mathit{variablename} \rangle$
$\langle \mathit{meta\_atom} \rangle$	::=	$\langle \mathit{metaname} \rangle$ | $\langle \mathit{variablename} \rangle$

Figure 3: EBNF for MD-SQL

Figure 3 shows an EBNF grammar for MD-SQL. MD-SQL is built on a relationally complete fragment of SQL that includes subqueries in the `from` and `where` clauses. Currently, MD-SQL does not support `group by` statements or aggregation, however it can be straightforwardly augmented with these capabilities. The data types $\langle \mathit{string} \rangle$, $\langle \mathit{real} \rangle$ and $\langle \mathit{integer} \rangle$ are compatible with their SQL counterparts. MD-SQL allows four types of variable declarations (whose syntax is shown in table 1): tuple variables, and *meta-variables* of type database, relation, or attribute.[4] As in SchemaSQL, we declare meta-variables using an arrow "`->`". Variable names and meta-names are character sequences starting with an upper case or lower case letter, respectively. Each component of the left hand sides of the declarations in table 1 is a *meta-atom*, i.e. a variable name or a meta-name.

Declaration Syntax	Variable Range
`-> D`	`D`: database names of federation
`D -> R`	`R`: relation names in database `D`
`D::R -> A`	`A`: attribute names of relation `R` of database `D`
`D::R T`	`T`: tuple variable over relation `R` of database `D`

Table 1: Variable declarations in MD-SQL.

[4]SchemaSQL allows a fifth type: *domain variables*, which are unnecessary in MD-SQL.

The main extensions to SQL that MD-SQL provides is an extended `AS` construct and a new `ALL` construct. The `AS` keyword serves to allow the user to name columns and relations. Columns are specified in the `SELECT` clause and named using an `AS` construct as usual. These column declarations may be nested within a second `AS`, which serves to name the *relations* in the output database. Relation path names of the form `dbName::relName` must be unique. If the same relation path name is given to two distinct relations in an MD-SQL query, a union is attempted. If the schema of the two relations is incompatible, this union will fail and the whole query returns $\emptyset_D$, the empty database having an empty schema. Similarly, if two distinct columns are given the same name, the MD-SQL query will return $\emptyset_R$, the empty relation having an empty schema. If several names are given for the same relation (or column), that relation (or column) is copied, and each copy is given one of the names.

The nested depth of parentheses in the `SELECT` clause syntactically determines whether an MD-SQL query returns a relation or database. We denote by MD-SQL$_{REL}$ those MD-SQL queries returning single relations and by MD-SQL$_{DB}$ those MD-SQL queries returning a database containing one or more relations. Note that the user is not forced to name the output of an MD-SQL query. This is mainly for convenience and compatibility with SQL. Since the user will not have access to this name later except through prior knowledge, the result of an MD-SQL query is given a unique internal name at run time. In the algebra (for the sake of formality), we assume database names are indeed given explicitly as strings by the user; however in MD-SQL such a demand is cumbersome and unnecessary.

As an example of the nested `AS` constructs, consider the MD-SQL query that restructures the information in the `univ-B` database into the format of `univ-A`:

```
SELECT ( T.category, B AS ‘dept’, T.B AS ‘avgSal’ ) AS ‘bToA’
FROM   univ-B::salInfo T, univ-B::salInfo -> B
WHERE  B != ‘category’
```

The meta-variable `B` ranges over attribute names in `univ-B::salInfo`, except for the attribute “`category`”. This query outputs a database containing one relation, “`bToA`”. Except for the second occurrence of the `AS` construct in the `SELECT` clause, it is the same as its SchemaSQL counterpart [8]. Note that the variable `B` has two distinct roles in this query. In the second column declaration in the `SELECT` clause, and in the `WHERE` clause, `B` refers to a *mention* of an attribute name (as a legitimate element of **dom** in its own right). However, in the construct `T.B`, `B` refers to a *use* of the attribute name, as a legitimate column name. This *use/mention distinction* is crucial for understanding the role of meta-variables in MD-SQL.

Another key extension to SQL, the `ALL` keyword in the `SELECT` clause, allows the user to specify a group of columns to be selected from a relation, whose names might not be explicitly known at compile time. The following two queries are equivalent:

```
SELECT ( ALL T.A ) AS ‘copyReln’
FROM   db::reln T, db::reln -> A
```

```
SELECT ( T.* ) AS ‘copyReln’
FROM   db::reln T
```

As another example, consider the translation from `univ-A` to `univ-C` in our sample federation. Data values must be promoted to relation names. In MD-SQL, this is done in two stages. First, copies of the `univ-A::salInfo` relation are generated, one for each value in the `dept` column of `univ-A::salInfo`. Finally, relations in this new database are rectified into the format of the `univ-C` database. Let the query `Q1` be as follows:

```
SELECT ( T.* ) AS S.dept
FROM   univ-A::salInfo T, univ-A::salInfo S
```

This query copies `univ-A::salInfo`. The translation from `univ-A` to `univ-C` is

```
SELECT  ( ALL T1.A1 ) AS R1
FROM    ( Q1 ) -> R1, ::R1 -> A1, ::R1 T1
WHERE   R1 = T1.dept AND A1 != 'dept'
```

This query relies heavily on the use/mention distinction. `R1` is declared as a meta-variable ranging over the relation names of the subquery `Q1`. `Q1` returns an unnamed database (hence no database name appears before the "`::`" in later references to the relation named by `R1`). `A1` is an attribute meta-variable ranging over the attributes of the relation named by `R1`; `T1` is a tuple variable ranging over the relation named by `R1`. In the latter two declarations, `R1` is being *used* to refer to the actual relation (that has a schema, tuples, etc); in the `SELECT` and `WHERE` clauses, the name indicated by `R1` is *mentioned*, behaving as a string for comparison and naming. The effect is that `R1` ranges over the copies of `univ-A::salInfo`, taking on the department names; tuples from `univ-A::salInfo` that match the current relation name are selected out, and projected on every attribute except the 'dept' attribute. The `ALL` in the `SELECT` clause is a *generalized projection* (§3), allowing the user to project a relation on multiple columns without having to know in advance what their names are. Column renaming is allowed in an `ALL` construct; details are given below (§3).

Finally, MD-SQL allows an `ALL` to appear in the `FROM` clause, modifying a subquery. The effect of the `ALL` is to join all relations in the database returned by the subquery and return the result as one relation.[5] As an example of this *generalized join*, consider translating the `univ-C` database into the format of `univ-B::salInfo`. The `ALL` construct forces a natural join, and we will not want the columns `avgSal` in the `univ-C` database to be joined. Thus the first step in the `univ-C` to `univ-B` translation is to rename the `avgSal` columns with the (distinct) relation names. Let `Q2` be the following query, where the inner `AS` construct drops the relation name into an attribute position.

```
SELECT ( T.category, T.avgSal AS R ) AS R
FROM    univ-C -> R, univ-C::R T
```

The translation from `univ-C` to `univ-B` is then given by

```
SELECT ( T2.* ) AS 'cToB'
FROM   ALL ( Q2 ) T2
```

[5]If the subquery returns a single relation, R, enclosing it in an `ALL` also returns R. If the subquery yields the empty database, $\emptyset_D$, enclosing it in an `ALL` returns the empty relation $\emptyset_R$.

Note that enclosing a subquery in the `FROM` clause by an `ALL` has the effect of returning a relation (not a database), so we can declare a tuple variable over the generalized join.

We have given three of the four translations indicated in figure 2, illustrated how data can be promoted to relation names, and shown how relation names and attribute names can be demoted to data. The final translation named in figure 2, `univ-A` to `univ-B`, is obtained by composing the A-to-C and C-to-B queries. The chief difference between ordinary SQL and MD-SQL is the `ALL` in the `SELECT` and `FROM` clause. Like SQL, MD-SQL queries are straightforwardly composed of simple components. It is this quality which yields a uniform translation into an algebraic form, and from there into a straightforward execution plan. In the next two sections, we present the meta-algebra and show it is equivalent to MD-SQL.

3 A Meta-Algebra of Names

Our meta-algebra is *two-tiered*: it is based on operations that are simple generalizations of relational operations and completed with a powerful *map* operator which will allow us to apply these operations to sets of named relations. As mentioned (§1), we assume an underlying set of domain elements, **dom**, from which both data values and meta-names are obtained. To illustrate the generalized relational operators, we will make use of the database `simpleDB` shown in figure 4.

r1:

att
b
d

r2:

att	a	b	c	d
a	1	2	3	4
c	4	3	2	1

Figure 4: `simpleDB`

Basic Expressions

We assume a distinguished subset of **dom** containing the elements **db**, **rel**, and **att** as well as their subscripted forms, such as $\mathbf{db}_1, \mathbf{rel}_{50}, \mathbf{att}_{101}$. These will be used to tag meta-data. Recall that relations are named by a *path expression* `dbName::relName` where `dbName` and `relName` $\in$ **dom**. This is syntactic sugar for a tuple of the form

$$\langle \mathbf{db} : \texttt{dbName}, \mathbf{rel} : \texttt{relName} \rangle \tag{1}$$

Relation variables[6] (X_1, R_3) and relation path names (`simpleDB::r1`) are basic expressions. A *meta-atom* is either a relation variable or a path name. The empty relation having an empty schema, $\emptyset_R$, is a basic expression.[7] Finally, constant relations of the form $\{\langle x : y \rangle\}$ where $x, y \in \mathbf{dom}$ are basic. We use 'a' as shorthand for the unary singleton relation $\{\langle a : a \rangle\}$ where $a \in \mathbf{dom}$. In addition, the following operations are basic:

1. *dbnames*(): returns a relation of sort $\{\langle \mathbf{db} : \mathbf{dom} \rangle\}$ containing names of federation databases.
2. Two operators for extracting relation pathnames:
   (a) $relnames_{names}(E)$, where E is a generalized relational expression yielding a column of database names. This operator returns a binary relation of sort $\{\langle \mathbf{db} : \mathbf{dom}, \mathbf{rel} :$

[6]These range over relation path names and can be instantiated using a *map* expression (see below).
[7]The result of any operation applied to $\emptyset_R$ always returns $\emptyset_R$.

$\mathbf{dom}\rangle\}$ containing the path names of relations in the databases named in E.

(b) $relnames_{db}(D)$, where D is a meta-algebra expression yielding a database. This operator returns a binary relation of sort $\{\langle \mathbf{db} : \mathbf{dom}, \mathbf{rel} : \mathbf{dom}\rangle\}$ containing the path names of relations in D.

The difference is that $relnames_{names}(E)$ operates on a unary relation of names, whereas $relnames_{db}(D)$ operates on an actual database. If E yields domain elements that do not actually name databases, $relnames_{names}(E)$ will not include any results for such names.

3. $name(M)$, where M is a meta-atom. If M is a path name `dbName::relName`, $name(M)$ returns the singleton relation $\{\langle \mathbf{db} : \texttt{dbName}, \mathbf{rel} : \texttt{relName}\rangle\}$. If M is a relation variable, $name(M)$ returns a relation containing the path name of the current instantiation of M.[8]

4. $schema(M)$, where M is a meta-atom. This operator returns a relation of sort $\{\langle \mathbf{db} : \mathbf{dom}, \mathbf{rel} : \mathbf{dom}, \mathbf{att} : \mathbf{dom}\rangle\}$ containing the attribute names of the relation M.

The algebra contains a relational sub-algebra, the *generalized relational expressions* (GREs). These expressions, together with a *map* operator and two database set operations, form the *meta-algebra expressions* (MAEs), a subset of which is distinguished as *meta-data queries* (MDQs). To begin the inductive definitions, each basic expressions is a GRE.

Subscripting

When translating an MD-SQL query into our meta-algebra, we will make use of a *subscripting* operator, $\rho(F, i)$, where F is any basic expression of the form $name(M)$, $schema(M)$, $relnames_{db}(D)$, $relnames_{names}(E)$, or $dbnames()$, and $i \in \mathbb{N}$ is fixed. For example,

$$\rho(name(simpleDB :: r1), 3) = \{\langle \mathbf{db}_3 : simpleDB, \mathbf{rel}_3 : r1\rangle\} \tag{2}$$

Generalized Projection

Let E be a GRE yielding a relation $R(A_1, ..., A_n)$ and U be a GRE yielding a unary singleton relation. Suppose U contains the constant $x \in \mathbf{dom}$ (as the value of its lone tuple). Then

$$\Pi_{*A_i(U)}(E) = \{t^{(x)} | \exists s \in E[s.A_i = B \text{ and } s.B = t.x]\} \tag{3}$$

Renaming (i.e. U) is not optional. The $*$ is a "dereference" operator on the attribute A_i. For each tuple it is not the A_i value that is projected but the value whose column name is given as the A_i value. We are potentially taking a different column from each tuple of E, so the result column must be given a new name. See table 2 for an example.

Furthermore, let E_1 be a GRE yielding a unary relation of sort $\{\langle \mathbf{att} : \mathbf{dom}\rangle\}$. Suppose E_1 yields a relation having values $x_1, ..., x_k \in \mathbf{dom}$. Let E_2 be a GRE yielding a k-ary relation of sort $\{\langle x_1 : \mathbf{dom}, ..., x_k : \mathbf{dom}\rangle\}$. Let E be a GRE yielding a relation. The column given by E_1 yields a list of attribute names which we will project E on. The relation E_2 gives the new name(s) for these columns. Suppose E_2 has tuples $\{\langle a_1^1, ..., a_k^1\rangle, ..., \langle a_1^m, ..., a_k^m\rangle\}$. Then

$$\Pi_{E_1(E_2)}(E) = \{t^{(a_j^i : 1 \le i \le m, 1 \le j \le k)} | \exists s \in E[s.x_j = t.a_j^i (1 \le i \le m, 1 \le j \le k)]\} \tag{4}$$

[8] Or the empty relation with the appropriate schema if M has no current instantiation.

<table>
<tr><td colspan="3">Basic Expressions</td></tr>
<tr><td colspan="3">R, ‘a’, $a \in \mathbf{dom}$; $\{\langle x : y\rangle\}$ where $x, y \in \mathbf{dom}$; $dbnames()$, $relnames_{names}(E)$, $relnames_{db}(D)$, $name(R)$, $schema(R)$</td></tr>
<tr><td colspan="3">Generalized Relational Algebra</td></tr>
<tr><td>Expression</td><td>Example(s)</td><td>Result</td></tr>
<tr><td>$\Pi_{*A(U)}(E)$</td><td>$\Pi_{*\texttt{att}(name(\texttt{simpleDB::r2}))}(\texttt{simpleDB :: r1})$</td><td><table><tr><td><code>r2</code></td></tr><tr><td>1</td></tr><tr><td>2</td></tr></table></td></tr>
<tr><td>$\Pi_{E_1(E_2)}(E)$</td><td>$\Pi_{\texttt{simpleDB::r1}(\{\langle b:x_1, d:y_1\rangle, \langle b:x_1, d:y_2\rangle\})}(\texttt{simpleDB :: r2})$</td><td><table><tr><td>$x_1$</td><td>$y_1$</td><td>$y_2$</td></tr><tr><td>2</td><td>4</td><td>4</td></tr><tr><td>3</td><td>1</td><td>1</td></tr></table></td></tr>
<tr><td>$\sigma_{t_1 \ \mathbf{op} \ t_2}(R)$</td><td>$\sigma_{*\texttt{att}=\{\langle \mathbf{int}:1\rangle\}}(\texttt{simpleDB :: r2})$</td><td><table><tr><td><code>att</code></td><td><code>a</code></td><td><code>b</code></td><td><code>c</code></td><td><code>d</code></td></tr><tr><td>a</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table></td></tr>
<tr><td>$\bowtie(D)$</td><td>$\bowtie(\{\texttt{simpleDB :: r1}, \texttt{simpleDB :: r2}\})$</td><td>$\texttt{simpleDB :: r1} \bowtie \texttt{simpleDB :: r2}$</td></tr>
<tr><td colspan="3">$E_1 \cup E_2$, $E_1 - E_2$, E_1 and E_2 are GREs.</td></tr>
<tr><td colspan="3">Database Set Operations</td></tr>
<tr><td colspan="3">$D_1 \cup D_2$, $D_1 - D_2$, D_1 and D_2 are meta-algebra expressions</td></tr>
<tr><td colspan="3">The Map Operator</td></tr>
<tr><td colspan="3">$map\Big(\lambda R_1 .....\lambda R_k(\Pi_C(E(R_1, ..., R_k; S_1, ..., S_m)); X_1, X_2(R_1), ..., X_k(R_1, ..., R_{k-1})); N\Big)$
$R_1, ..., R_k$ relation variables; $S_1, ..., S_m$ GREs;
E a GRE, N a distinguished column of E, C a list of projection conditions</td></tr>
</table>

Table 2: Summary and examples of meta-algebra.

This is *generalized projection (with renaming)*. We project E on several columns simultaneously and E_2 gives (via the correct attribute names) the new name(s) for these columns. Note that if a column is given multiple names we copy it under each new name. If two distinct columns are given the same name, the operation will fail, returning $\emptyset_R$. If some attribute x_l does not appear in the schema of E, of course no such column will appear in the output. A special case of this is ordinary projection: $\Pi_{\{\langle \mathbf{att}:A_i\rangle\}}(E) = \Pi_{A_i}(E)$. Note that column renaming is optional in a generalized projection. An example of a generalized projection is given in table 2. As in the relational algebra, the composition of two or more projection operations is abbreviated to $\Pi_C(E)$ where C is a list of projection conditions.

Generalized Selection

Given E, a GRE yielding a relation $R(A_1, ..., A_n)$ we allow selections of the form $\sigma_{t_1 \ \mathbf{op} \ t_2}(E)$ where $\mathbf{op} \in \{<, >, =, \neq, \leq, \geq\}$, $t_i \in \{*A_i, A_i, U\}$, and U is a GRE yielding a unary singleton relation.[9] Expressions of the form $\sigma_{A_i \ \mathbf{op} \ A_j}(E)$ or $\sigma_{A_i \ \mathbf{op} \ `a'}$ have the usual meaning. The

[9] For consistency, we can suppose integers are represented by unary relations of the form $\{\langle \mathbf{int} : i\rangle\}$

only new selection expressions are those involving the dereference operator $*$, for example:

$$\sigma_{*A_i \ \mathbf{op}\ *A_j}(E) = \{t^{(A_1,\ldots,A_n)} | t \in E \text{ and } \exists B_1, B_2 [t.A_i = B_1 \text{ and } t.A_j = B_2 \text{ and } t.B_1 \ \mathbf{op}\ t.B_2\} \quad (5)$$

Another example is given in table 2. As in the relational algebra, the composition of two or more selection operations is abbreviated $\sigma_C(E)$ where C is a list of selection conditions.

Generalized Join
Let D be a meta-algebra expression yielding a database containing the relations $r_1, r_2, ..., r_k$. Then the *generalized join* is given as

$$\bowtie (D) = r_1 \bowtie r_2 \bowtie ... \bowtie r_k \quad (6)$$

If the r_i have no column names in common, this produces the full cross product of the r_i. We define $\bowtie (\emptyset_D) = \emptyset_R$.

Set Operations on Relations
Let E_1, E_2 be GREs. Expressions of the form $E_1 - E_2$ and $E_1 \cup E_2$ are GREs, assuming E_1, E_2 yield relations of the same schema, returning $\emptyset_R$ in the event of a schema mismatch.

We can now state formally what a *generalized relational expression* (GRE) is.

Definition 1 *Every basic expression is a GRE. Furthermore, if E, F, E_1, E_2 are GREs of the appropriate type and C, B are lists of projection and selection conditions respectively, then $\Pi_C(E)$, $\sigma_B(E)$, $\rho(F, i)$, $E_1 \cup E_2$ and $E_1 - E_2$ are GREs. If D is an MAE (below), then $\bowtie (D)$ is a GRE.*

Every GRE yields a relation. If a GRE has free relation variables, its result is $\emptyset_R$. We define the *generalized relational queries* (GRQs) as GREs with no free variables. A GRQ yields an *unnamed* relation, as do relational algebra queries. In order to give free variables instantiations and every relation a name, we define a *map* operator which applies a GRE to a set of relations, outputting a (named) database as a set of named relations.

The map Operator
Let $R_1, ..., R_k$ be relation variables and $X_1, ..., X_k$ GREs yielding relation path names. Let E be a well-formed GRE and C a list of well-formed projection expressions. Let N be either the name of a column of E or a domain element and `outputDB` $\in$ **dom**. Then the result of

$$map\big(\lambda R_1 \cdots \lambda R_k [\Pi_C(E); X_1, ..., X_k]; N; \text{‘}\texttt{outputDB}\text{’}\big) \quad (7)$$

is a database named “`outputDB`,” created according to the algorithm in figure 5, where $\emptyset_D$ is the empty database (having an empty schema). Let *outputDB′* be a unique name, synthesized in order to avoid conflicts with existing databases. At the end of the algorithm, the database `outputDB` created (or overwritten) with the contents of *outputDB′*.

The column N may be projected out during step 8; this is why the outer Π_C is not absorbed into E. In steps 8 and 13, tuples are inserted into a named relation in *outputDB′*. If this named relation already has contents (from a previous instantiation of $R_1, ..., R_k$) a union is attempted. If the schema of the existing contents is incompatible with the new

initialize $outputDB' := \emptyset_D$;
for each instantiation of $\langle R_1, ..., R_k \rangle$ to a vector of names $\langle r_1, ..., r_k \rangle$ where $r_i \in X_i$ for $1 \leq i \leq k$ **do**
  create the relation yielded by E according to the current instantiation;
  if N names a column of E **do**
    for each value $v \in \Pi_N(E)$
      if there is no relation named v already in $outputDB'$
        create an empty relation of correct schema in $outputDB'$ whose name is v;
      insert $\Pi_C(\sigma_{N=v}(E))$ into relation v of $outputDB'$ (fail if schema mismatch);
  else { In this case $N \in$ **dom** }
    if $outputDB' = \emptyset_D$
      create relation N of correct schema in $outputDB'$;
    else
      insert $\Pi_C(E)$ into relation N of $outputDB'$ (fail if schema mismatch);
$outputDB := outputDB'$;
return $outputDB$;

Figure 5: The *map* algorithm.

contents, the whole *map* operation fails and returns $\emptyset_D$. Also, if E has extraneous free variables, the result of the map expression will be $\emptyset_D$. We assume the relation variables are instantiated in order, $R_1, R_2, ...$ Each X_i may depend (recursively) in the instantiations of $R_1, ..., R_{i-1}$. We do not allow circular references; any map expression containing such outputs $\emptyset_D$.

Definition 2 *A* map *expression of the above form is a* meta-algebra expression *(MAE). Furthermore, if D_1 and D_2 are meta-algebra expressions, so are $D_1 \cup D_2$ and $D_1 - D_2$.*

Note E (the expression inside the *map* operator) is a GRE; however, $relnames_{db}(D)$ takes an MAE as argument. Thus E can contain a meta-algebra expression as a subexpression, enclosed by the $relnames_{db}$ operator, as can the naming expressions X_i ($1 \leq i \leq k$). A GRE is not an MAE, nor is an MAE a GRE; however, a GRE can refer to the result of an MAE via the names returned by $relnames_{db}$. This is the motivation for calling the GREs an "algebra on names." Finally, we can formally characterize meta-data queries:

Definition 3 *A* meta-data query *(MDQ) is a meta-algebra expression with no free variables.*

In the next section we prove the main result of this paper, namely that the meta-algebra queries are exactly the MD-SQL queries. In fact, we will see GRQ + MDQ $\equiv$ MD-SQL.

4 Equivalence of MD-SQL and the Meta-Algebra

In this section, we establish the main result of this paper:

Theorem 1 *GRQ + MDQ $\equiv$ MD-SQL$_{REL}$ + MD-SQL$_{DB}$.*

As a corollary, we will develop a normal form for meta-algebra expressions which will aid in designing a non-intrusive MD-SQL query engine (§6).

4.1 Meta-Algebra to MD-SQL

In order to prove the result, we need to show how every GRQ has an equivalent MD-SQL$_{REL}$ counterpart. In fact, each GRQ has a natural MDQ counterpart. For example, a path name of the form `dbName::relName` corresponds to

$$map(\lambda R.[R; \{\texttt{dbName} :: \texttt{relName}\}]; \text{`}\texttt{relName}\text{'}; \text{`}\texttt{dbName}\text{'}) \tag{8}$$

The effect of the *map* operation is to force one to give the result of the GRQ a proper path name. As discussed, we do not force MD-SQL users to do this.

Lemma 1 *GRQ* $\equiv$ *MD-SQL$_{REL}$*

We will prove every GRQ has an MD-SQL$_{REL}$ counterpart; the converse proceeds similarly to the proof in §4.2.

Basic Expressions. Most of the basic expressions translate straightforwardly into MD-SQL. For example, $schema(\texttt{dbName} :: \texttt{relName})$ translates to

```
SELECT 'dbName' AS 'db', 'relName' AS 'rel', A AS 'att'
FROM   dbName::relName -> A
```

Note that we can simply indicate the distinguished domain elements **db**, **rel**, and **att**. We will see a procedure below for translating MDQs into MD-SQL$_{DB}$. Suppose D is an MDQ yielding database `outputDB` that translates into an MD-SQL$_{DB}$ query `Qd`. Then $relnames_{db}(D)$ translates to the following query, where `R` ranges over relation names in `outputDB`:

```
SELECT 'outputDB' AS 'db', R as 'rel'
FROM   ( Qd ) -> R
```

Projections. Let E be a GRQ corresponding to MD-SQL$_{REL}$ query `Qe`. Then $\Pi_{*x(\text{`}y\text{'})}(E)$ is

```
SELECT T.A as 'y'
FROM   ( Qe ) T,  ( Qe ) -> A
WHERE  T.x = A
```

Now suppose E is as above, X_1 yields a column of attribute names and X_2 is a renaming relation (§3). Suppose that X_1 and X_2 translate to MD-SQL$_{REL}$ queries `Qx1` and `Qx2` respectively. Then $\Pi_{X_1(X_2)}(E)$ translates to

```
SELECT ALL T.A as S.A
FROM   ( Qx2 ) S, ( Qe ) T, ( Qe ) -> A
WHERE  EXISTS  ( SELECT X1.*
                 FROM   ( Qx1 ) X1
                 WHERE  X1.att = A )
```

Recall that both MD-SQL and the meta-algebra use a *join semantics*. In this case, the relations in the `FROM` clause of the above query will be joined. However, we do not want X_2 and E to be joined (this would essentially intersect them) unless columns of X_2 are mentioned elsewhere (for example in the `WHERE` clause). Thus, in the special case of renaming relations, they are not included in the join created by the `FROM` clause. Such renaming relations are clearly coupled (the attribute meta-variable A *must* be the same on the left and right hand sides of the `AS` keyword), and can be omitted appropriately without problems.

Selections. As above, suppose E is a GRQ that translates to a query `Qe`. The selection $\sigma_{t_1 \mathbf{op} t_2}$ translates straightforwardly. For example, $\sigma_{*a=*b}(E)$ translates to the following query:

```
SELECT T.*
FROM   ( Qe ) T, ( Qe ) -> A, ( Qe ) -> B
WHERE  T.a = A AND T.b = B AND A = B
```

Subscripting, Union and Set Difference. The subscripting operator may be imitated directly in MD-SQL by simply renaming columns using subscripted versions of the distinguished domain elements. Additionally, if E_1 and E_2 are GRQs, then $E_1 \cup E_2$ is obtained simply by unioning their respective MD-SQL$_{REL}$ counterparts via the `UNION` operation of MD-SQL. Set difference can be obtained using a `NOT EXISTS` construction in the `WHERE` clause, as in the translation from the ordinary relational algebra to SQL.

Generalized join. Let D be an MDQ having MD-SQL$_{DB}$ counterpart `Qd`. Then the generalized join $\bowtie (E)$ is simply given by

```
SELECT T.*
FROM   ALL ( Qd ) T
```

This completes our proof of lemma 1. $\square$.

Finally, to establish that every query in GRQ+MDQ has an MD-SQL counterpart, we will show every MDQ query can be translated into MD-SQL$_{DB}$. Consider the *map* operator. Let $R_1, ..., R_k$ be relation variables and C a list of projection conditions. Let E be a GRQ with corresponding MD-SQL query `Qe`. Let $X_1, ..., X_k$ be GRQs yielding relations of sort $\{\langle \mathbf{db} : \mathbf{dom}, \mathbf{rel} : \mathbf{dom} \rangle\}$ with corresponding MD-SQL queries `Qx1`, ..., `Qxk`. Consider

$$map\big(\lambda R_1 \cdots \lambda R_k[\Pi_C(E); X_1, ..., X_k]; N; \text{‘}\texttt{outDB}\text{’}\big) \tag{9}$$

Let `Cprime` be the translation of the projection conditions into the format of the MD-SQL `SELECT` clause (following techniques described in the translation of GRQs to MD-SQL$_{REL}$). This *map* expression translates to the following MD-SQL$_{DB}$ query:

```
SELECT ( Cprime ) AS Nprime
FROM   -> D1, D1 -> R1, D1::R1 T1, D1::R1 -> A1,
       ...,
       -> Dk, Dk -> Rk, Dk::Rk Tk, Dk::Rk -> Ak,
       ... [ other declarations of E, for example any
             general join or fixed relations ]
WHERE  EXISTS ( SELECT S1.*
                FROM ( Qx1 ) S1
                WHERE S1.db = D1 AND S1.rel = R1 )
```

```
AND ... AND
EXISTS ( SELECT Sk.*
         FROM ( Qxk ) Sk
         WHERE Sk.db = Dk AND Sk.rel = Rk )
AND  [ conditions for any selections in E ]
```

The variable (or domain element) that names the result relations, `Nprime`, is derived from the name specifier N as follows. Suppose N names a column of E. If N names a column arising from a call to one of the meta-data operators ($name(\cdot)$, $schema(\cdot)$, etc), it takes the form **db**, **rel**, **att** or one of their subscripted versions. For example, if N names the **att** column of a call to $schema(R_i)$, we will translate N to `Ai` in the MD-SQL query below (i is a fixed integer, $1 \leq i \leq k$). Otherwise (if N names a column not generated by one of the meta-data extractors), we use a tuple variable ranging over the relation it belongs to in order to specify the correct column. For example, the query translating univ-A::salInfo into a database of relations in the format of university C uses the `dept` column of `univ-A::salInfo` to name the result relations (§2).

A union of *map* expressions can be obtained using MD-SQL's `UNION` facility. A set difference of such can be imitated using a `NOT EXISTS` construction. This completes our proof that MDQs can be translated into MD-SQL$_{DB}$.

4.2 MD-SQL to Meta-Algebra

As a base case, we will illustrate the translation method from MD-SQL to the meta-algebra on a (reasonably complex) query having no subqueries. We will then discuss how to translate MD-SQL queries with arbitrarily nested subqueries. The query we will translate is:

```
SELECT ( ALL T1.A1,
         T2.A2 AS 'colName',
         T3.xxx  AS  yyy )  AS R2
FROM   -> D1, D1 -> R1, D1::R1 T1, D1::R1 -> A1,
       -> D2, D2 -> R2, D2::R2 T2, D2::R2 -> A2,
       -> D3, D3 -> R3, D3::R3 T3, D3::R3 -> A3
WHERE  A1 != D2 AND T2.foo = R2
```

First, calculate the naming relations: $X_i = relnames_{names}(dbnames())$ for $1 \leq i \leq 3$. Any conditions involving only these relations can appear here (for example `R2 != 'xyz'`).

In SQL, the `FROM` clause translates into one large cross product; here, it translates into one large natural join. Meta-variables are introduced by our meta-data extractor operators. The number of relation meta-variables in the `FROM` clause determines how many relation variables will appear in the *map* expression; in this case there are three: R_1, R_2, R_3. Let

$$E = \rho(name(R_1), 1) \bowtie R_1 \bowtie \rho(schema(R_1), 1) \bowtie \rho(name(R_2), 2) \bowtie R_2 \bowtie \rho(schema(R_2), 2) \bowtie \rho(name(R_3), 3) \bowtie R_3 \bowtie \rho(schema(R_3), 3) \quad (10)$$

Let "$\texttt{outputDB}$" $\in$ **dom**. Let $E' = \sigma_{\mathbf{att}_1 \neq \mathbf{db}_2 \wedge foo \neq \mathbf{rel}_2}(E)$. Then a meta-algebra expression corresponding to this query is:

$$map\big(\lambda R_1 \lambda R_2 \lambda R_3 [\Pi_{\Pi_{\mathbf{att}_1}(E'), *\mathbf{att_2}(\text{'colName'}), xxx(\text{'yyy'})}(E'); X_1, X_2, X_3]; \mathbf{rel}_2; \text{'}\texttt{outputDB}\text{'}\big) \tag{11}$$

Translating MD-SQL subqueries. There are several cases to consider.

1. Subqueries in the `FROM` clause.
   (a) If the subquery returns a database over which a relation meta-variable is defined, say R_j, the subquery will appear as a map expression enclosed by the $relnames_{db}$ operator in the naming relation X_j corresponding to R_j. Note that variables used in the subquery that are declared in an outer scope will appear as recursive dependencies, i.e. X_j may depend on $R_1, ..., R_{j-1}$. Alternatively, if the subquery returns a database qualified by an `ALL` construct, the subquery will appear as a *map* expression within a generalized join inside the outer *map*.
   (b) If the subquery returns a relation, it corresponds to a GRE which will appear as a join component inside the *map* expression.
2. Subqueries in the `WHERE` clause.
   (a) Subqueries in the `WHERE` clause enclosed by an `EXISTS` construct can be moved into outer queries. Their variables can be included in the join expression arising within the top-level `FROM` clause and their conditions translate into selections on this join. If any conditions result in an empty result in the `EXISTS` clause, this will be reflected in an empty join once the `EXISTS` subquery has been moved into the outer scope.
   (b) Subqueries in the `WHERE` clause enclosed by a `NOT EXISTS` construct are slightly trickier. The idea is generally the same as in ordinary SQL. Suppose there are p `NOT EXISTS` clauses in a certain query. We will first union the results of these `NOT EXISTS` clauses into one relation having the correct schema which can then be subtracted from the relation created in the query (as if there were no `NOT EXISTS` clauses).

Subqueries consisting of unions will translate to unions of map expressions, concluding our proof of theorem 1. □

As a corollary, we have the following normal form. Every MDQ can be written as a finite union of expressions of the form

$$map\big(\lambda R_1 \cdots \lambda R_k [\Pi_{C_1}(\sigma_{D_1}(J_1) - \Pi_{C_2}(\sigma_{D_2}(J_2))); X_1, X_2(R_1), ..., X_k(R_1, ..., R_k)]; N; \text{'a'}\big) \tag{12}$$

In the above, J_1 and J_2 are join expressions over $R_1, ..., R_k$, possibly some other fixed relations, and including some generalized joins (whose arguments may contain other, nested *map* expressions). The projection Π_{C_2} ensures that what is subtracted (i.e. the results of any `NOT EXISTS` clauses) has the proper schema.

5 Theoretic Results

The main result of this section is that MD-SQL defines exactly PSPACE computations. The proof of this is sketched in §5.1, below. In §5.2, we briefly state some results concerning when the output type of an MD-SQL query can be determined at compile time.

5.1 Complexity Results

We begin by showing how an NP-complete problem, *three coloring*, can be translated into MD-SQL. The input database, `triColor`, is exhibited in figure 6. This is a representation of the graph $G = (V, E)$ where $V = \{v_1, ..., v_n\}$ and E is given by the `edges` relation. The goal is an MD-SQL query that returns 'yes' if a valid three-coloring of the graph $G = (V, E)$ exists and 'no' otherwise. First, we will create all possible colorings of G using a technique similar to that involved in transforming `univ-A` into `univ-B` (§2). In query `Q1` (below), we copy the relation `vertices` $\times$ `colors` n times, naming each copy after a vertex.

`vertices:`

vertex
v_1
$\vdots$
v_n

`colors:`

color
r
g
b

`edges:`

source	sink
v_i	v_j
$\vdots$	$\vdots$

Figure 6: `triColor` database.

```
SELECT  (U.vertex, V.color) AS W.vertex
FROM    triColor::vertices U, triColor::colors V, triColor::vertices W
```

`Q1` returns a database having n copies of the `vertices` relation, appropriately named. Now, query `Q2` (following) first drops the names of the relations created in `Q1` into the `color` column and projects out each `vertex` column. Then, the database resulting from the subquery is joined, returning a single relation encoding all possible colorings of the n vertices:

```
SELECT T.*
FROM   ALL ( SELECT ( R.color AS R ) AS R
             FROM   ( Q1 ) -> R           ) T
```

Now we must remove from the relation in `Q2` those colorings that violate the three-coloring property, namely that no two vertices sharing an edge have the same color. This is achieved using a straightforward `NOT EXISTS` clause in the following query:

```
SELECT T1.*
FROM   ( Q3 ) T1, ( Q3 ) -> A, ( Q3 ) -> B
WHERE  NOT EXISTS ( SELECT T2.*
                    FROM   triColor::edges T2
                    WHERE  T2.source = A AND T2.sink = B
                           AND T1.A = T1.B )
```

The attribute meta-variables `A` and `B` behave as iterators ranging over the n vertices. The solution to the three-coloring problem can then be obtained by using another query which checks for existence and outputs 'yes' or 'no' accordingly.

It is the generalized join which gives MD-SQL (and the meta-algebra) its power. Without the `ALL` in the `FROM` clause, MD-SQL is in PTIME (in fact LOGSPACE). Furthermore, this phenomenon is not limited to MD-SQL. Assuming the operation of the *unfold* operator specified in [10], a direct counterpart of this three-color solution can be phrased in SchemaSQL, as can the proof of the following proposition:

Proposition 1 *NP $\subset$ MD-SQL. In fact, PH $\subset$ MD-SQL.*

The proof of proposition 1 is based on a characterization of NP as existential second order sentences given by Fagin [3]. We assume a finite domain, D, over which to interpret (first order) formulae and several relations $S_1, ..., S_k$ over D which make up our input "database". NP is characterized as sentences of the following form:

$$F := \exists R_1^{l_1} \cdots \exists R_m^{l_m} \phi(R_1, ..., R_m; S_1, ..., S_k) \tag{13}$$

where ϕ is a first order sentence (therefore expressible in MD-SQL). Given $M = (D; S_1, ..., S_k)$ and F, we will show that there exists an $\hat{M}$ and $\hat{F}$ such that (i) $M \models F$ iff $\hat{M} \models \hat{F}$ and (ii) $\hat{F}$ is expressible in MD-SQL in the context of $\hat{M}$. The argument is reasonably straightforward. We need an MD-SQL query that (as in the case of three-coloring) generates all possible relation candidates for $R_1^{l_1}, ..., R_m^{l_m}$ and then we use ϕ enclosed in a `NOT EXISTS` clause to weed out inappropriate relations. The catch is that we cannot generate all possible relation candidates of arbitrary length schemas. However, it is well known that every relation of arity l can be expressed as l binary relations representing tuple IDs and associated values. Each of the l relations gives the l-th column of the original relation. We use a similar technique to translate F and M into $\hat{F}$ and $\hat{M}$. We illustrate the technique on a simplified formula:

$$F := \exists R^l \phi(R; S_1, ..., S_k) \tag{14}$$

Let $\hat{D} = D^l = \{e_1, e_2, ..., e_{|D^l|}\}$. Let 0, 1 be new elements (not previously in D). Then we can create in MD-SQL a relation of the form exhibited in figure 7. Each tuple is a bit vector representing a set of tuple IDs (the e_i behave as our tuple IDs). What we seek is the existence of l binary relations $E_R^1, ..., E_R^l$ such that for $e \in \hat{D}, d \in D$, $\langle e, d \rangle \in E_R^i$ iff $e \in R$ and d is the i-th component of e. Let $\hat{M} = (\hat{D} \cup \{0, 1\}; S_1, ...S_k)$ and

e_1	e_2	$\cdots$	$e_{\lvert\hat{D}\rvert}$
0	0	$\cdots$	0
1	0	$\cdots$	0
$\vdots$			

Figure 7: All possible relations.

$$\hat{F} := \exists E_R^1 \cdots \exists E_R^l \hat{\phi}(E_R^1, ..., E_R^l; S_1, ..., S_k) \tag{15}$$

where $\hat{\phi}$ is ϕ translated appropriately to mention the E_R^i instead of R. It should be clear that $M \models F$ iff $\hat{M} \models \hat{F}$ and that $\hat{F}$ is expressible in MD-SQL using a similar technique as in the three-coloring problem (above). The case of a general F of the form in equation 13 is similar. Furthermore, the translation holds in the case of alternating $\forall$ and $\exists$ higher-order quantifiers; hence it should be clear that the entire polynomial hierarchy, $PH \subset$ MD-SQL, as claimed. A limitation of this result is that the transformation from M and F to $\hat{M}$ and $\hat{F}$ cannot itself be expressed in MD-SQL without a tupling construct. $\Box$

Additionally, we have the following result:

Proposition 2 *PSPACE = MD-SQL*

Proof sketch: It should be clear that MD-SQL can be implemented in PSPACE. Furthermore, PSPACE can be characterized as SO(FO+TC) [5]. FO+TC yields formulas of the form

$$TC(\phi(x_1, ..., x_k; y_1, ..., y_k; z_1, ..., z_m)) \tag{16}$$

where TC is a transitive closure predicate relative to the vectors $\overrightarrow{x} = x_1, ..., x_k$ and $\overrightarrow{y} = y_1, ..., y_k$. The strategy in translating such a formula into MD-SQL is to first make $\hat{\phi}$ that just refers to tuple IDs (similar to the proof that $PH \subset$ MD-SQL), compute the transitive closure of $\hat{\phi}$ in MD-SQL, then translate back (manually). The transitive closure of binary relations can be expressed in MD-SQL (proof omitted). $\Box$

The upshot is that meta-data queries are inherently expensive. Everything in MD-SQL can be imitated by SchemaSQL or SchemaLog; hence these languages cannot be considered *in general* to be efficient, even if most naturally occuring queries have efficient evaluations. The generalized join of our meta-algebra is inherently expensive, however it is hard to see how arbitrary schemas can be created from the data at run time without such an operation. In other words, it is hard to see how allowing the transparent migration of data to meta-data (and vice versa) can be anything other than inherently powerful. It is an open problem whether there is a feasible, natural subfragment of the meta-algebra that allows untyped output but where the size of intermediate relations is polynomially bounded. One candidate is MD-SQL minus the `ALL` in the `FROM` clause; several of the queries in §2 remain expressible, however the `univ-A` to `univ-B` translation (for example) is no longer expressible.

5.2 Typedness

In general, an MD-SQL query is untyped. That is, the schema of the output database (or relation) cannot be decided at compile time.[10] This result follows directly from the fact that it is undecidable whether a first-order formula on any input returns an empty database. If we leave out the `ALL` in the `SELECT` clause, as well as constructions of the form "`T.*`", then one obtains a typed fragment of MD-SQL. This fragment still has the potential for high expressiveness because of the remaining `ALL` in the `FROM` clause. On the other hand, changing MD-SQL slightly leads to a typed language. We include a check with each query that asks "is my type the following?" If the check succeeds, the query will proceed normally; if the check fails, we will simply create an empty database (or relation) of the appropriate schema. This is a rather contrived way of ensuring MD-SQL is typed at compile time.

[10]We can decide at compile time whether an MD-SQL query outputs a database or a relation, of course.

6 Implementation

An implementation of a fragment of MD-SQL without the `ALL` in the `SELECT` and `WHERE` clauses and not supporting subqueries exists [13]. We are currently completing an implementation of an MD-SQL engine that processes queries based on the EBNF in figure 3. The system is highly portable and can be run on top of any DBMS that supports the Dynamic-SQL/CLI standard [2].

First, we compile an MD-SQL query using Lex and Yacc [11] into instances of C++ objects (a parse tree). Using this parse tree, instantiations for the meta-variables in the query are generated in the order in which they are declared in the `FROM` clause. This may involve recursively creating result tables corresponding to subqueries.

All tables generated during the evaluation of an MD-SQL query are temporary (an MD-SQL query is assumed to have no side effects). Furthermore, all generated tables are given unique, synthesized names (separate from any name the user assigns), to avoid gratuitous name clashes.

Given a particular instantiation of the variables in the input query, we rely on the fact that every MD-SQL query corresponds directly to an MDQ in normal form (§4.2). This normal form is evaluated according to the *map* algorithm already discussed (§3). Once every possible instantiation has been considered, the final result is output and all intermediately generated tables are removed. Optimization of MD-SQL query evaluation has two facets: (i) relational queries are dispatched to the underlying DBMS which will optimize them, and (ii) certain optimizations (for example of `WHERE` clause conditions that contain reference only to meta-variables) can be performed by our MD-SQL engine.

7 Related Works

Meta-data query languages had their origin with HILOG [1], a complex object based logic programming language with function terms. There is a single namespace over which terms are defined; these terms can appear in predicate and attribute positions. One limitation of HILOG is that terms have fixed arity, hence limiting the flexibility needed for relational meta-querying. For example, the query from `univ-A` to `univ-B` is impossible to express since the schema width of the `univ-B` counterpart depends on the input data.

SchemaLog [7], like HILOG, is a logic programming language for meta-querying, however it operates in a relational setting. The key contribution of SchemaLog is to overcome the fixed arity problem of HILOG: a tuple is seen as a function and SchemaLog's syntax allows attribute meta-variables to range over tuple values. This allows the width of output relations to vary with the input, hence the `univ-A` to `univ-B` is expressible in SchemaLog.

Other work on meta-data query languages was done by Ross [14] and Jain *et al* [6]. Ross introduced an algebra and calculus wherein it is possible to demote relation names to ordinary domain values (although not to promote data to relation names). A safe fragment of his calculus is proceduralized through equivalence to his algebra. Another approach is given by the *uniform model* of [6]. Here, schemas, tuples and relations are all encoded in one database consisting of binary relations. An algebra and equivalent calculus are introduced, and the inherent expressibility of relational query languages that allow varying width output schema is suggested.

Throughout the paper, we have indicated how MD-SQL is similar to (and inspired by) SchemaSQL. SchemaSQL is, in turn, a derivative of SchemaLog [8]. The "Extended Algebra" underpinning a fragment of SchemaLog contains operators similar to our basic GRE expressions [9]. The tabular algebra-like operators *unite*, *split*, *unfold* and *fold* underpinning SchemaSQL [10] can be obtained from our meta-algebra; for an indication of how this is achieved see the example translations in §2. The *unfold* operator as described in [10] is as powerful as our generalized join, unlike the original *unfold* operator of [4] which can be implemented in polynomial time.

8 Conclusions

In this paper, we have introduced a relation-based language for meta-data querying, MD-SQL. MD-SQL has a uniform design in that restructuring and untyped queries are directly supported without recourse to a `create view` statement. MD-SQL extends SQL in a natural way. The most salient differences between MD-SQL and SQL arise from the `ALL` in the `SELECT` and `FROM` clauses. The `ALL` in the `SELECT` clause allows the creation of output relations with varying schema; the nested `AS` constructs furthermore allow the output of a varying number of relations. The `ALL` in the `FROM` clause allows the user to join a varying number of relations, as opposed to a fixed number in SQL.

We have proved an equivalence between MD-SQL and a relation-based algebra. This provides (i) a clear semantics for MD-SQL and (ii) the basis for an implementation of MD-SQL over an existing relational engine. Our implementation can benefit from known optimizations applying to relational algebra expressions.

Our complexity results clearly show how relational meta-query languages that support untyped output are inherently powerful. In particular, we demonstrated that MD-SQL can express every PSPACE query. Thus in general, one cannot expect efficient support for all meta-data queries. An open problem is to isolate an interesting fragment of MD-SQL that can be implemented in PTIME, but does support untyped queries. A candidate is MD-SQL minus the `ALL` in the `FROM` clause. Another open problem is to identify the fragment of MD-SQL that results in typed queries. A natural candidate is MD-SQL minus the `ALL` in the `SELECT` clause. The component-wise design of MD-SQL and the availability of an equivalent algebra facilitates the study of such questions.

Acknowledgments

The authors extend their thanks to Laks Lakshmanan for discussing and sending an advance copy of his forthcoming VLDB paper [10]. Thanks also go to Jan Paredaens for helpful comments concerning the precursor of the meta-algebra.

References

[1] Chen W, M. Kifer and D.S. Warren. HILOG: A Foundation for Higher Order Logic Programming. *Journal of Logic Programming*, 15(3): 187-230, 1993.

[2] Date, C.J. and Hugh Darwen. *A Guide to the SQL Standard*, 4th Ed. Addison-Wesley, 1997.

[3] Fagin, Ronald. Monadic generalized spectra. *Zeitschrift für Math. Logik und Grund. der Math*, 21, pp. 123-134, 1975.

[4] Gyssens, Marc, L.V.S. Lakshmanan and S.N. Subramanian. Tables as a paradigm for querying and restructuring. In *Proc. ACM Symp. on PODS*, 93-103 June 1996.

[5] Immerman, Neil. *Descriptive Complexity*. Springer-Verlag, 1998.

[6] Jain, Manoj, Anurag Mendhekar and Dirk Van Gucht. A Uniform Data Model for Relational Data and Meta-Data Query Processing. *COMAD*, 1995.

[7] Lakshmanan, L.V.S, F. Sadri and I.N. Subramanian. On the Logical Foundations of Schema Integration and Evolution in Heterogeneous Database Systems. *DOOD* '93, 81-100. LNCS 760, December 1993.

[8] Lakshmanan L.V.S, F. Sadri and I.N. Subramanian. SchemaSQL – A Language for Interoperability in Multi-Database Systems. *VLDB* '96, 239-250, Bombay, India, September 1996.

[9] Lakshmanan L.V.S, F. Sadri and I.N. Subramanian. Logic and Algebraic Languages for Interoperability in Multidatabase Systems. *Journal of Logic Programming*, 33(2): 101-149, November, 1997.

[10] Lakshmanan L.V.S, F. Sadri, and I.N. Subramanian. On an Efficient Implementation of SchemaSQL. *VLDB* '99, Edinburgh, Scotland, September 1999. (Forthcoming.)

[11] Levine, John R, Tony Mason and Doug Brown. *Lex and Yacc*. O'Reilly and Associates, Inc, 1995.

[12] Miller, R.J. Using Schematically Heterogeneous Structures. *ACM SIGMOD* '98, 189-200, Seattle, WA, May 1998.

[13] Rood, C.M. An Implementation of a Subset of SchemaSQL. On the web at `http://www.php.indiana.edu/~crood`.

[14] Ross, Kenneth. Relations with relation names as arguments: Algebra and calculus. In *Proc. of the ACM Symp. on PODS*, 346-353. San Diego, June 1992.