Augmenting SQL with Dynamic Restructuring to Support Interoperability in a Relational Federation

Catharine Wyss[1], Felix Wyss[2], and Dirk Van Gucht[1]

[1] Indiana University, Computer Science Dept.
{crood, vgucht}@cs.indiana.edu
[2] Interactive Intelligence, Indianapolis, IN
felixw@inin.com

Abstract. In this work, we consider augmenting SQL with constructs allowing *dynamic restructuring*. Dynamic restructuring occurs naturally in a framework including second-order capabilities for integrating metadata and data within single queries. The context and motivating example for this extension is a federation of relational databases, containing semantically similar information in schematically disparate formats. In ordinary SQL, the output schema for a given query is always known. In order to support interoperability within such a federation, it is desirable to add to SQL the ability to dynamically restructure information among databases, based on the data and metadata of the input relations. Based on a motivating example, we define a hierarchy for characterizing dynamic restructuring capabilities. We then describe a language kernel, the Meta-Query Language (MQL), for dynamically restructuring relational data. MQL includes only simple extensions of SQL but captures almost all of the hierarchy of dynamic restructuring capabilities. We then investigate two extensions of this kernel for obtaining complete data/metadata integration. One extension (MQL+ON) remains LOGSPACE while the other (MQL+JOINALL) captures the PSPACE queries. We conclude by comparing these two extensions and stating the open problem of developing a declarative extension of MQL that captures PTIME and retains completeness with respect to our dynamic restructuring hierarchy.

1 Introduction

Interoperability among heterogeneous information sources has become a major research issue within the last decade. Many information sources consist of relational database management systems (RDBMS). Interoperability issues affect relational databases as well; thus, a sub-problem of the interoperability problem for heterogeneous information sources is that of interoperating among RDBMSs. Even within a single organization, data from disparate relational sources must be integrated. Solutions to the wider problem of integrating heterogeneous data sources (relational and non-relational) are often unsuitable or unwieldy for restructuring relational data, and may require database administrators (DBAs) and users having to learn entirely new query languages and paradigms. In contrast, we desire a language for interoperability that is *downward compatible* with SQL, supporting the portability of legacy code. Ideally, the language should share some of the key features of SQL, such as ease of query formulation, sufficient expressiveness, a modest complexity (LOGSPACE) and support for aggregation [11]. Such

an interoperable query language for federated RDBMSs would be an important first step toward general interoperability. Furthermore, such a language could provide sophisticated support to federation DBAs, allowing the creation of efficient metadatabase front ends, and assisting in the definition and implementation of wrappers, mediators, and integrating views.

1.1 A Motivating Example

Our work investigates issues that arise when adding interoperability extensions to ordinary SQL. The context of these extensions is a federation of RDBMSs where constituent databases store semantically similar information in schematically disparate formats. As a motivating example, consider the `consultants` federation (figure 1), illustrating varying methods of storing payroll information about consultants. John and Sally have each consulted for 4 different firms, and similar information is stored within these firms. Firm `A` stores salary information in a single relation, `firm-A::payInfo`. Firm `B` uses a single relation as well (`firm-B::payInfo`), however consultant names appear as attributes in this relation. Firm `C` uses several relations, each named after a consultant (the relations `firm-C::john` and `firm-C::sally`). Firm `D` uses a separate database for each consultant; information within these databases is stored in the relation `firmDprojs`.

Our goal is to add to SQL the ability to dynamically restructure information between any two relations in such a federation, *in real time*. This ability would greatly facilitate schema integration and provide a general purpose framework for federation administrators or sophisticated users seeking support for interoperability within the federation.

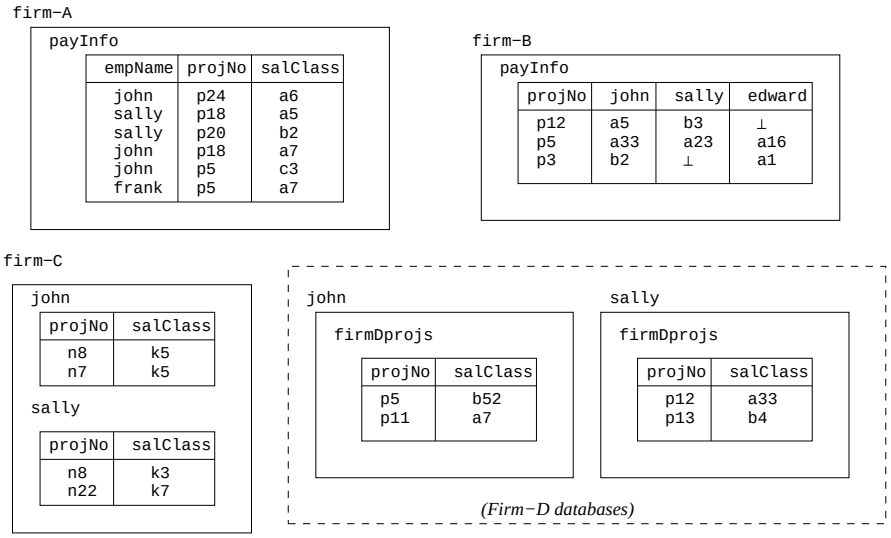

Fig. 1. The `consultants` Federation.

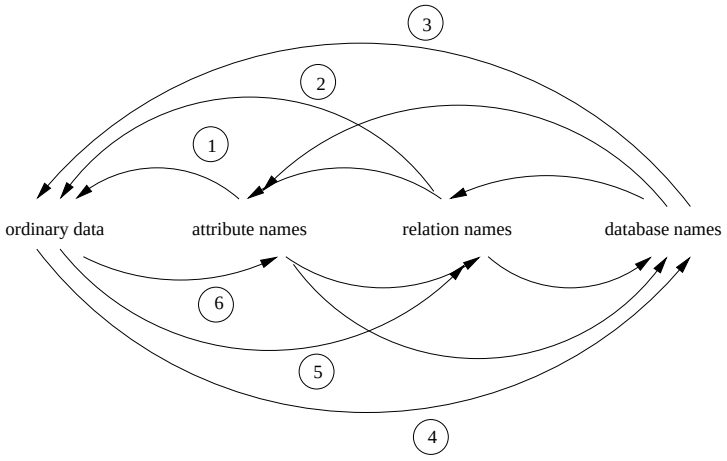

Fig. 2. Transformations between data and metadata.

The `consultants` federation suggests a standard for measuring the success of relational interoperability in a language. Figure 2 illustrates the twelve possible transformations between data and metadata that arise when restructuring data between formats in the `consultants` federation. We identify six of these transformations (numbered 1-6) as a complete, minimal generating set (*i.e.* using the six transformations in composition, all twelve can be achieved). We have stratified the six generating transformations into three *levels of interoperability*, corresponding roughly to the increasing complexity of a language that achieves the transformations.

Level I – *metadata integration*. At this level, the language supports the ability to demote metadata (database names, relation names, attribute names) to data (transformations 1-3).
Level II – *pathname creation*. At this level, the language supports level I plus the additional ability to promote data to relation pathnames (transformations 4 and 5), which consist of a database name prepended to a relation name. [1]
Level III – *horizontal restructuring*. At this level, the language supports level I and II restructuring, and additionally the ability to promote data to attribute names (transformation 6). This transformation is the most difficult and unnatural to achieve in SQL, since the output of an SQL query must be of a fixed type which is known in advance. Horizontal restructuring demands that the type of the output be calculated *dynamically*, at run-time.

We term query languages that support level I and above capabilities *metaquery languages*. The ability to declare metavariables ranging over database, relation and attribute names gives the language a second-order syntax (thus producing "meta" queries).

[1] We use the notation `dbName::relName` for relation pathnames, following the SchemaSQL convention [11].

RDBMSs come equipped with built-in data dictionaries, which means that we have easy access to the federation's metadata: database names, relation names, and attribute names. We assume that metadata and data are taken from the same domain of atomic elements (called **dom**).[2] To simplify presentation, we assume all elements of **dom** are alphanumeric strings beginning with a lowercase letter.

The idea of adding second order capabilities to relational languages is not new. The next section gives a brief history of query languages which fall within our metaquery hierarchy.

1.2 Related Work

One of the first higher-order query languages is HiLog [1]. HiLog is 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, supporting metadata querying. One of the major limitations of HiLog is that terms have fixed arity, hence dynamic output schemas cannot be generated using a HiLog query.

Soon after the introduction of HiLog, the Ross Algebra [16] and SchemaLog [10, 12] were introduced. The Ross Algebra allows relation names to be demoted to ordinary domain values. However, the Ross Algebra supports only limited metadata querying and does not allow the schemas of output relations to vary dynamically with the contents of the input relations. In contrast, SchemaLog is a logic programming language allowing dynamically typed output schemas and full data/metadata integration. The syntax of SchemaLog is similar to the Interoperable Database Language developed in [9]. SchemaLog has two drawbacks: (1) it is not declarative and is thus foreign to many users, and (2) SchemaLog uses tuple identifiers and supports full recursion, a significant drawback for efficient query processing over large databases. SchemaSQL evolved from SchemaLog to meet the need for declarative SQL-like metadata query languages. However, SchemaSQL did not retain the completeness of SchemaLog: only one column of values can become metadata in a restructuring view statement [5, 11].

Solution frameworks for interoperability within federated information systems contain potential for metaquerying. However, in most such frameworks, component schemas have to be known in advance of specifying restructuring mappings (for example, this is the case in the procedural mapping language BRIITY [7]).

Several distinctions are commonly made with reference to federated information systems (FIS) architectures, such as monolithic versus hierarchical [13], centralized versus distributed [13], and coupled (loosely or tightly) versus autonomous [17]. Furthermore, a distinction between multidatabases (decentralized, autonomous relational sources) and federated (schema integrated) databases is often made [14, 18, 19]. We do not view the MQL/MA framework as restricted to a single architecture; rather our platform could be useful for administrators and sophisticated users of any FIS architecture where some degree of global data interoperation is sought among relational databases. Many existing tools for FIS architectures are based on object models. We feel the suc-

[2] The exception is the null element $\perp$, which can appear as data but *not* metadata.

cess of the relational platform calls for an approach based more closely on Codd's relational framework; early precedents for such an approach include [2].

Most RDBMSs support dynamic restructuring, if they support the Dynamic SQL standard [3]. Dynamic SQL embeds SQL within a procedural language and allows SQL queries to be constructed and executed at run-time. However, these facilities do not have a solid theoretical base within the relational model itself.

2 MQL: A Core Language for Dynamic Restructuring

In this section, we introduce the MQL core. MQL allows the declaration of *metavariables* in the `FROM` clause, which dynamically range over input database, relation, and attribute names.[3] Note that the declaration of metavariables places the MQL core at least at level I of our restructuring hierarchy.

2.1 MQL Syntax

Figure 5 shows the EBNF syntax of the MQL kernel. Variable names begin with *uppercase* letters and atoms (elements of **dom**) begin with lowercase. Strings are domain elements enclosed in single quotes. Note that subqueries may appear in the `FROM` clause.

```
SELECT T.empName AS 'empName' INTO 'p5_consultants' WITHIN 'resDB'
FROM   firm-A::payInfo AS T
WHERE  T.projNo = 'p5'
```

Fig. 3. Q1: consultants on `p5`

```
SELECT B AS 'empName', T.projNo AS 'projNo', T.B AS 'salClass'
         INTO 'payInfo' WITHIN 'b2a'
FROM   firm-B::payInfo -> B, firm-B::payInfo AS T
WHERE  B <> 'projNo'
```

Fig. 4. Q2: `firm-B::payInfo` $\mapsto$ `b2a::payInfo`

Examples of MQL queries appear in figures 3 and 4. Query Q1 selects the names of those consultants from the `firm-A::payInfo` relation that worked on project `p5` and places the result in a federation containing a single relation (`p5_consultants`) within a single database (`resDB`). Query Q2 restructures the information in relation `firm-B::payInfo` into the format of `firm-A::payInfo`, placing the result in relation `payInfo` of database `b2a`.

MQL queries specify named relations and databases into which output tuples are placed. This is done in the `SELECT` clause using the `INTO` and `WITHIN` keywords. Note

[3] Although the syntax of MQL is similar to that of SchemaSQL [11], the semantics differs in that range declarations appearing in the same MQL query are *independent* from one another, as in SQL (§2.2).

⟨*query*⟩ ::= **SELECT** ⟨*col_decls*⟩
INTO ⟨*name_term*⟩
WITHIN ⟨*name_term*⟩
FROM ⟨*variable_decl*⟩ {**,** ⟨*variable_decl*⟩}*
[**WHERE** ⟨*condition*⟩ {**AND** ⟨*condition*⟩}*]
| **(** ⟨*query*⟩ **) UNION (** ⟨*query*⟩ **)**
| **(** ⟨*query*⟩ **) MINUS (** ⟨*query*⟩ **)**

⟨*col_decls*⟩ ::= ⟨*col_decl*⟩ {**,** ⟨*col_decl*⟩}*

⟨*col_decl*⟩ ::= ⟨*name_term*⟩ **AS** ⟨*string*⟩

⟨*variable_decl*⟩ ::= **−>** ⟨*varname*(*db*)⟩
| **(** ⟨*query*⟩ **) −>** ⟨*varname*(*db*)⟩
| ⟨*meta_atom*(*db*)⟩ **−>** ⟨*varname*(*rel*)⟩
| ⟨*meta_atom*(*db*)⟩ **::** ⟨*meta_atom*(*rel*)⟩ **−>** ⟨*varname*(*att*)⟩
| ⟨*meta_atom*(*db*)⟩ **::** ⟨*meta_atom*(*rel*)⟩ **AS** ⟨*varname*(*tup*)⟩

⟨*condition*⟩ ::= **(** ⟨*condition*⟩ **)**
| ⟨*name_term*⟩ ⟨*cond_operator*⟩ ⟨*name_term*⟩
| [**NOT**] **EXISTS (** ⟨*query*⟩ **)**

⟨*cond_operator*⟩ ::= **=** | **!=** | **<=** | **<** | **>** | **>=**

⟨*name_term*⟩ ::= ⟨*string*⟩ | ⟨*varname*(*tup*)⟩**.**⟨*meta_atom*(*att*)⟩ | ⟨*varname*(_)⟩

⟨*meta_atom*(*X*)⟩ ::= ⟨*metaname*⟩ | ⟨*varname*(*X*)⟩

⟨*metaname*⟩ ::= **a-z** {**(a-z** | **A-Z** | **0-9** | **−** | **_)**}*

⟨*varname*(*X*)⟩ ::= **A-Z** {**(a-z** | **A-Z** | **0-9** | **−** | **_)**}*

Fig. 5. EBNF for core MQL.

that variables may appear in the name terms specifying the output relation and database names; thus MQL provides the ability to promote data to relation and database names dynamically (level II of our restructuring hierarchy). As an example, figure 6 gives an MQL query that restructures the `firm-A` database into the `firm-D` databases.

```
SELECT T.projNo AS projNo, T.salClass AS salClass
         INTO 'firmDprojs' WITHIN T.empName
FROM   firm-A::payInfo AS T
```

Fig. 6. `firm-A` $\mapsto$ `firm-D`.

The query in figure 7 gives a more obvious example of the power of core MQL. This query returns the pathnames of relations having an attribute in common, in relations named after the common attribute.

```
SELECT D1 AS 'dbName', R1 AS 'relName' INTO A1 WITHIN 'resDB'
FROM   D1::R1 -> A1, D2::R2 -> A2
WHERE  A1 = A2
```

Fig. 7. Relations with a common attribute.

2.2 MQL Semantics

MQL query evaluation utilizes a standard *nested loops* algorithm. The loop instantiates all metavariables as suitable elements of **dom** , depending on the form of the declaration these variables appear in. For example, a declaration of the form `firm-A::R1 -> A1` would specify that `R1` is interpreted as the name of a relation in the `firm-A` database and `A1` is interpreted as the name of an attribute in the schema of this relation. Tuple variable declarations may depend crucially on metavariables; for example, a declaration of the form `D2::salInfo AS T2` says that `T2` ranges over tuples of relations named `salInfo` in any federation database. Figure 8 summarizes the semantics of core MQL.

FOR database meta-atoms D_1, D_2, ..., D_I ranging over domain elements
d_1^1, ..., $d_1^{i_1}$; d_2^1, ..., $d_2^{i_2}$; ...; d_I^1, ..., $d_I^{i_I}$; respectively;
FOR relation meta-atoms R_1, R_2, ..., R_J ranging over domain elements
r_1^1, ..., $r_1^{j_1}$; r_2^1, ..., $r_2^{j_2}$; ...; r_J^1, ..., $r_J^{j_J}$; respectively;
FOR attribute metavariables A_1, A_2, ..., A_K ranging over domain
elements a_1^1, ..., $a_1^{k_1}$; a_2^1, ..., $a_2^{k_2}$; ...; a_K^1, ..., $a_K^{k_K}$; respectively;
FOR tuple variables T_1, T_2, ..., T_M ranging over relations R_1^1, ..., $R_1^{m_1}$;
R_2^1, ..., $R_2^{m_2}$; ...; R_M^1, ..., $R_M^{m_M}$; respectively;
IF the WHERE clause is satisfied when the domain values are
substituted for database, relation and attribute metavariables
and values from appropriate tuples are substituted for all tuple
attribute references
THEN
evaluate the name terms in the SELECT clause according to
the substituted domain and tuple values and produce the tuple
of values that results into the resulting relation named within
the resulting database named.

Fig. 8. Nested loops algorithm for evaluating flat MQL queries.

In addition, we have developed a complete algebratization of MQL [20]. The equivalent algebra (the *Meta-Algebra*, or MA) is a principled extension of the Relational Algebra (RA) including operators for dereferencing metanames and extracting metadata from the federation dictionary. The translation to this algebra fundamentally parallels the translation of SQL to the RA; thus many known techniques for query evaluation and optimization carry over to the MQL/MA framework. Like SQL, MQL is LOGSPACE; therefore we anticipate an efficient implementation allowing real-time restructuring within a relational federation.

2.3 Limitations of MQL

Although the extensions to SQL that MQL involves are quite straightforward, the MQL core already achieves level II of our metaquery hierarchy (figure 2). However, the MQL core contains no facility for promoting data to attribute names (level III restructuring). The next two sections augment the MQL core to achieve level III metaqueries.

3 MQL+ON: "Transposing" A Relation

Our first language providing level III metaqueries is obtained by adding an `ON` construct to the `SELECT` clause of MQL. The syntax of the `ON` construct is identical to that of the existing `AS` construct except that the term to the right of the keyword may now contain variables. The `ON` keyword signifies that values arising on its right are to be expanded *horizontally* into columns as a dynamic output schema and coupled tuple-by-tuple with values arising to the left of the keyword. Thus, we must enforce (semantically) that left and right results have the same "shape" (for example, two columns from the same relation).

This type of horizontal restructuring is conceptually similar to matrix transposition, in that a column of values becomes a row of attribute names. The contents of the columns named to the left of the `ON` keyword line up under these new attribute names, so that values that were "beneath" one another are now "beside" one another. As an example, a query restructuring `firm-A::payInfo` into `firm-B::payInfo` is given in figure 9. The tuple-by-tuple semantics of this query is depicted in figure 10.

```
SELECT ((A.projNo AS 'projNo', A.salClass ON A.empName)
         INTO 'payInfo') INTO 'a2b'
FROM   firm-A::payInfo AS T
```

Fig. 9. `firm-A::payInfo` $\mapsto$ `a2b::payInfo`.

For the query in figure 9, the output schema is:

$$\{v : v = \texttt{projNo}\} \cup \{v : v \in \texttt{firm-A::payInfo.empName}\}.$$

Tuples in the output relation, s, arise from tuples $t \in$ `firm-A::payInfo`, according to the rule:

$$s[X] = \begin{cases} t[\texttt{salClass}] \text{ when } X = t[\texttt{empName}] \\ \perp \text{ otherwise.} \end{cases}$$

As an further explication of the semantics of our "transpose" operation, we can interpret the query in figure 9 according to our nested loops semantics (figure 8). Each interpretation of the tuple variable `T` corresponds to one tuple in the input relation, `firm-A::payInfo` (thus there are six distinct output tuples, s_0 through s_5. The output federation is determined in two stages. First, the output schema is determined (which is {`projNo`,`john`,`sally`,`frank`}); then, the output data is determined. Note that, in general, each distinct interpretation of the meta- and tuple variables in an MQL (and MQL+ON) query yields exactly one tuple in the output federation.

3.1 The Complexity of MQL+ON

In this section, we state a result concerning the expressiveness of MQL+ON.

Proposition 1. *MQL+ON* $\subseteq$ LOGSPACE.

	firm-A::payInfo				a2b::payInfo			
	empName	projNo	salClass		projNo	john	sally	frank
$t_0 : \mathtt{T} \rightarrow$	john	p24	a6	$\mapsto s_0 :$	p24	a6	$\perp$	$\perp$
$t_1 : \mathtt{T} \rightarrow$	sally	p18	a5	$\mapsto s_1 :$	p18	$\perp$	a5	$\perp$
$t_2 : \mathtt{T} \rightarrow$	sally	p20	b2	$\mapsto s_2 :$	p20	$\perp$	b2	$\perp$
$t_3 : \mathtt{T} \rightarrow$	john	p18	a7	$\mapsto s_3 :$	p18	a7	$\perp$	$\perp$
$t_4 : \mathtt{T} \rightarrow$	john	p5	c3	$\mapsto s_4 :$	p5	c3	$\perp$	$\perp$
$t_5 : \mathtt{T} \rightarrow$	frank	p5	a7	$\mapsto s_5 :$	p5	$\perp$	$\perp$	a7

Fig. 10. Variable Interpretations to Output Tuples.

Corollary 1. *MQL* $\subseteq$ LOGSPACE.

The heart of the proof of proposition 1 relies on the tuple-by-tuple evaluation of MQL (and MQL+ON) expressions. The only workspace that is necessary are indices into the input federation: the output is generated a tuple at a time, from the input.

All of the transformations among databases in the `consultants` federation are possible in MQL+ON. An open question is whether this level of expressibility is generally sufficient for interoperability among relational databases. One direction for answering this question might involve investigating the parallels between MQL+ON and the *tabular algebra* [6], since the tabular algebra has been shown to be complete for transformations involving tabular data.

4 MQL+JOINALL: Joining Arbitrary Sets of Relations

In this section, we consider an alternative construct for creating dynamic output schemas in MQL; this second language is based on the MQL core (figure 5) plus a new `FROM` clause construct using the `JOINALL` keyword. MQL+JOINALL is obtained by adding the ability to declare tuple variables over outer joins of relations returned by a query. Thus, we can fold up an entire federation of relations into a single relation (the outer join of the constituent relations) and declare a tuple variable over the result. In some ways, the idea of generalizing the outer join to accept a varying number of relations as argument is more natural than an operation like matrix transposition, since relations do not naturally admit any coupling between distinct columns (unlike the data in a matrix). The language resulting from adding the generalized join is more powerful than MQL+ON and can imitate all the transformations possible in MQL+ON, and any PSPACE transformations in general (§4.2).

4.1 MQL+JOINALL Syntax and Semantics

As noted, MQL+JOINALL allows the `JOINALL` keyword to appear in the `FROM` clause, modifying a subquery. The effect of the `JOINALL` is to join all relations in the federation returned by the subquery and return the result as one relation. As an example of this *generalized join*, consider translating the `firm-C` database into the format of `firm-B::payInfo`. The `JOINALL` construct results in a natural outer join, however we

do not want the salClass columns in the firm-C relations to be joined. Thus the first step in the firm-C to firm-B translation is to rename the salClass columns with their (distinct) relation names. Let Q1 be the following query, where the AS construct in the SELECT clause (from MQL) drops the relation name into the salClass attribute position. The result of query Q1 is depicted in figure 11 (a).

```
SELECT T.projNo AS 'projNo', T.salClass AS R INTO R WITHIN 'tmpDB'
FROM    firm-C -> R, firm-C::R AS T
```

The translation from firm-C to firm-B is then given by the following query (result depicted in figure 11 (b).

```
SELECT (( T2.* ) INTO 'payInfo') INTO 'c2B'
FROM   JOINALL ( Q1 ) AS T2
```

Enclosing the subquery Q1 in the JOINALL keyword returns a relation that is the natural join of all the relations in the tmpDB database. The schema of this relation is the union of the schemas of the relations in tmpDB. The contents of the join relation are obtained from the component relations (adding $\perp$ values where necessary). Note that enclosing a subquery in the FROM clause by a JOINALL has the effect of returning a relation (not a federation), so we can declare a tuple variable over the generalized join.

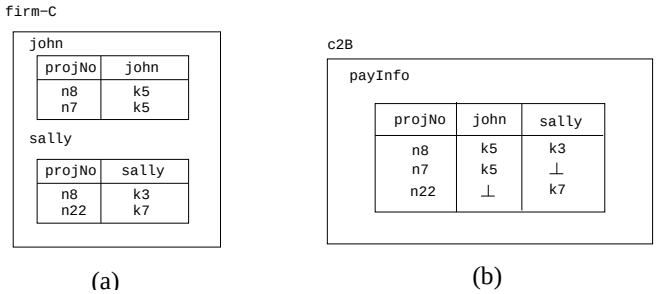

Fig. 11. Result of query Q1 and translation from firm-C to firm-B format.

4.2 The Complexity of MQL+ALL

The main result of this section is that MQL+ALL exactly captures PSPACE. To illustrate the power of MQL+JOINALL, we first show how an NP-complete problem, *three coloring*, can be stated. The input database, triColor, is exhibited in figure 12. 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 MQL+JOINALL query that returns 'yes' if a

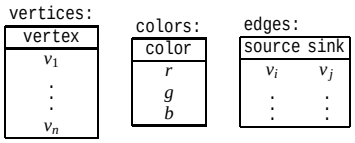

Fig. 12. triColor database.

valid three-coloring of the graph $G = (V,E)$ exists and 'no' otherwise. First, we will create all possible colorings of G. In query Q2 (below), we copy the relation vertices $\times$ colors n times, naming each copy after a different vertex.

```
SELECT  U.vertex AS 'vertex', V.color AS 'color' INTO W.vertex
          WITHIN 'colorings1'
FROM    triColor::vertices AS U, triColor::colors AS V,
        triColor::vertices AS W
```

Now, query Q3 (following) drops the names of the relations created in Q2 into the color column and projects out each vertex column. This results in n relations, each containing a single column named after a distinct vertex, representing a single coloring of all n vertices. Then, the database resulting from the subquery is joined, returning a single relation encoding all possible colorings of the n vertices. Note that joining the n relations creates the 3^n possible colorings since we have manipulated all column names to be distinct – this ensures the join is in fact a complete Cartesian product.

```
SELECT T.* INTO 'all_possible' WITHIN 'colorings3'
FROM   JOINALL ( SELECT R.color AS R INTO R WITHIN 'colorings2'
                 FROM   ( Q2 ) -> R )  AS T
```

Now we must remove from the relation in Q3 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.* INTO 'violators' WITHIN 'colorings4'
FROM   ( Q3 ) AS T1, ( Q3 ) -> A, ( Q3 ) -> B
WHERE  NOT EXISTS ( SELECT T2.*
                    FROM   triColor::edges AS 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 the colorings4::violators relation for existence and outputs 'yes' or 'no' accordingly.

It is the generalized join which gives MQL+JOINALL its power. Without it, MQL is restricted to LOGSPACE queries (corollary 1). Proposition 2 summarizes the previous example.

Proposition 2. *NP $\subset$ MQL+JOINALL. In fact, PH $\subset$ MQL+JOINALL.*

The proof of proposition 2 is based on a characterization of NP as existential second order sentences given by Fagin [4]. For the full proof, see [15].

Additionally, we have the following result:

Proposition 3. *PSPACE = MQL+JOINALL*

The proof of this result uses the fact that PSPACE can be characterized as SO(FO+TC) [8]. See [15] for details.

The upshot of propositions 2 and 3 is that queries involving a generalized join are inherently expensive.

5 A Comparison of MQL+ON and MQL+JOINALL

Each of the new operators, transpose and generalized join, naturally tackles only one of the restructurings possible in the `consultants` federation. For the transpose, restructuring from `firm-A::payInfo` to `firm-B::payinfo` becomes easy (see the query in figure 9 – this corresponds to transformation 6 in our metaquery hierarchy in figure 2). For the generalized join, the most natural translation is rather from the `firm-C` relations to the `firm-B::payInfo` relation (relation names become attribute names). When performing other restructurings in each language, recourse to the pure MQL core is necessary. For example, to restructure from the `firm-C` relations to `firm-B::payInfo` in MQL+ON, we first translate the `firm-C` relations into the format of `firm-A::payInfo` (figure 13) so that we may apply the `firm-A::payInfo` to `firm-B::payInfo` transformation.

```
SELECT R1 AS 'empName', T2.projNo AS 'projNo', T2.salClass AS 'salClass'
          INTO 'c2aReln' WITHIN 'c2aDB'
FROM   firm-C -> R1, firm-C::R2 AS T2
WHERE  R1 = R2
```

Fig. 13. `firm-C` to `firm-A::payInfo` in MQL+ON.

Conversely, when restructuring from `firm-A::payInfo` to `firm-B::payInfo` using MQL+JOINALL, we first write a query restructuring `firm-A::payInfo` into the format of the `firm-C` database (figure 14). The transformation from the `firm-C` relations to `firm-B::payInfo` can then be applied to complete the circle.

```
SELECT T.projNo AS 'projNo', T.salClass AS 'salClass'
          INTO T.empName WITHIN 'a2cDB'
FROM   firm-A::payInfo AS T
```

Fig. 14. `firm-A::payInfo` to `firm-C` in MQL+JOINALL.

Thus, each new operator can be said to be more "natural" for some transformations than others. As noted, it is an open problem whether the transformations indicated by the `consultants` example prove sufficient in general practice. Since MQL+ON is restricted to LOGSPACE, for example, the transitive closure cannot be formulated in MQL+ON. This discounts many natural queries asking for relationships between federation elements. On the other hand, although we can formulate such queries in MQL+JOINALL, the added complexity of the language is a problem for database processing. Even though an ordinary user would not be expected to (for example) solve three-coloring using MQL+JOINALL, it would be easy to unintentionally write infeasible queries, resulting in unexplained behavior of the system.

We summarize our work in the next section.

6 Conclusions

The relational model remains a key paradigm in information storage. Any interoperability framework should contain the ability to naturally restructure relational data. Restructuring relational data dynamically involves the ability to create dynamic output schemas and vary ranges based on input metadata, capabilities which are lacking in standard SQL. We have developed a core language, MQL, which allows metavariables in the `FROM` clause and provides a general framework for logical mappings from federations to federations. We presented two extensions of core MQL for creating coupled horizontal output schemas: MQL+ON and MQL+JOINALL. The language MQL+ON remains LOGSPACE, like SQL, and allows transformations among the databases in federations similar to `consultants` (figure 1). However, the "transposition" effect of the `ON` construct is somewhat opaque and it seems likely that there are desirable transformations that cannot be effected in LOGSPACE. In contrast, the language MQL+JOINALL relies on the concept of a generalized join for creating horizontal output schemas. The generalized join is more natural than the transpose within a relational framework, since relations do not involve coupling between distinct columns (as a matrix-type transpose operation requires). However, MQL+JOINALL is equivalent to the PSPACE queries, which is a problem for efficient query processing. MQL+JOINALL effectively allows metavariables to become limited iterators, resulting in complex behavior. It seems that there should be a declarative language between MQL+ON and MQL+JOINALL, that allows the creation of coupled horizontal output schemas yet remains feasible, for example capturing the PTIME queries. As of now, this language remains undiscovered.

7 Acknowledgments

Several individuals contributed to our research, including Jan Paredaens, Mehmet Dalkilic and Dennis Groth. This version was immeasurably improved during conversations with Edward Robertson, Chris Giannella and the IU Database Lab. We also thank the anonymous reviewers for helpful suggestions and comments.

References

1. Widong Chen, Michael Kifer, and David S. Warren. HiLog: A foundation for higher-order logic programming. Technical report, Computer Science Department, SUNY at Stony Brook, 1990.
2. E.F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, 1979.
3. C. J. Date and Hugh Darwin. *A Guide to the SQL Standard*. Addison-Wesley, 1997.
4. Ronald Fagin. Monadic generalized spectra. *Zeitschrift für Math. Logik und Grund. der Math*, 21:123–134, 1975.
5. Frédéric Gingras and Laks V.S. Lakshmanan. nD-SQL: A multi-dimensional language for interoperability and OLAP. In *Proceedings of the 24th VLDB Conference*, 1998.
6. Marc Gyssens, Laks V.S. Lakshmanan, and Iyer N. Subramanian. Tables as a paradigm for querying and restructuring. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS '96)*, 1996.

7. Theo Härder, Günter Sauter, and Joachim Thomas. The intrinsic problems of structural heterogeneity and an approach to their solution. *VLDB Journal*, 8(1):25–43, 1999.
8. Niel Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.
9. Ravi Krishnamurthy, Witold Litwin, and William Kent. Language features for interoperability of databases with schematic discrepancies. In *Proceedings of the 1991 ACM SIGMOD*, 1991.
10. Laks V.S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous database systems. In *DOOD '93*, 1993.
11. Laks V.S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. SchemaSQL – a language for interoperability in relational multi-database systems. In *Proceedings of the 22nd VLDB Conference*, 1996.
12. Laks V.S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. Logic and algebraic languages for interoperability in multidatabase systems. *Journal of Logic Programming*, 32(2):101–149, 1997.
13. Sharad Mehrotra, Henry F. Korth, and Avi Silberschatz. An architecture for large multidatabase systems. Technical Report CS-TR-92-50, Department of Computer Science, University of Texas at Austin, 1993.
14. Shamkant B. Navathe and Michael J. Donahoo. Towards intelligent integration of heterogeneous information sources. In *Proceedings of the 6th International Hong Kong Computer Society Database Workshop*, 1995.
15. Catharine M. Rood, Dirk Van Gucht, and Felix I. Wyss. MD-SQL: A language for meta-data queries over relational databases. Technical Report TR-528, Computer Science Department, Indiana University, 1999.
16. Kenneth A. Ross. Relations with relation names as arguments: Algebra and calculus. In *Proceedings of the 11th ACM Conference on Principles of Database Systems (PODS '92)*, 1992.
17. Amit P. Sheth and Vipul Kashyap. So far (schematically) yet so near (semantically). In *Proceedings of the IFIP DS-5 Conference on Semantics of Interoperable Database Systems*, pages 283–312, 1992.
18. Markus Tresch and Marc H. Scholl. A classification of multi-database languages. Technical Report 94-07, University of Ulm Faculty of Computer Science, 1994.
19. Mark W. W. Vermeer and Peter M. G. Apers. On the applicability of schema integration techniques to database interoperation. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 179–194, 1996.
20. Catharine Wyss and Dirk Van Gucht. A relational algebra for data/metadata integration in a federated database system. In *Proceedings of the 10th annual Conference on Information and Knowledge Management (CIKM '01)*, 2001.