Extending Relational Query Optimization to Dynamic Schemas for Information Integration in Multidatabases

Catharine M. Wyss
Indiana University
cmw@cs.indiana.edu

Felix I. Wyss
Interactive Intelligence, Inc.
felixw@inin.com

ABSTRACT

This paper extends relational processing and optimization to the FISQL/FIRA languages for dynamic schema queries over multidatabases. Dynamic schema queries involve the creation and restructuring of metadata at run time. We present a full implementation of a FISQL/FIRA engine based on our work, which includes subqueries and all transformational capabilities of FISQL/FIRA on distributed, multidatabase platforms. An important application of the system is to enhance traditional information architectures by enabling the creation and maintenance of dynamic wrappers and mapping queries at source databases within GAV, LAV, GLAV, peer-to-peer, or other integration frameworks. In addition to fully supporting FISQL/FIRA on multidatabases, our implementation introduces a bi-level optimization paradigm where purely relational sub-fragments of queries are pushed into source engines. This paradigm shares features of canonical distributed database processing, but has a new dimension through the extension of the relational model to dynamic schemas. We present empirical results showing the feasibility of optimization in this context, and discuss tradeoffs involved. Our system is the first to extend relational databases with these capabilities on this scale.

1. INTRODUCTION

Data integration is a perennial issue in database management, and continues to receive much attention in recent years [12, 13]. Although the relational model is widespread and an elegant data storage paradigm, to date it has lacked features necessary for large scale data integration [19]. In particular, a problem that plagues data integration systems is that of *dynamic schemas*, such as those in our example federation of Figure 1. Until recently, the relational model has lacked the *transformational completeness* to express the data-metadata restructuring inherent in such applications.

Figure 1 shows a loosely coupled federation of databases for a retail supply network. In this federation, there is no single, global schema, and source databases are updated independently of one another. As shown, relation ORDERS of the CA database contains aggregated total sale information in a normalized relational format. On the other hand, relation ORDERS of the IL database shows semantically equivalent data in a "spreadsheet" format, where each customer occupies a separate column. In database NY, a separate relation is used for each customer. Note that in this example there is overlap among the databases, for example the same information about customer SMITH is represented in all three databases. On the other hand, customer MILLS is only represented in the IL database.

Our goal was to implement a relational database system that could effectively and dynamically query data and metadata in federations such as Figure 1. Recently, we developed a relational language for dynamic schema querying, *Federated Interoperable SQL* (FISQL) [19]. The main advantage of FISQL in terms of query processing and optimization purposes is that it has an equivalent algebra, *Federated Interoperable RA* (FIRA), that is a direct extension of canonical RA. In particular, FIRA inherits the algebraic properties of RA that are essential for query optimization, such as commutativity, associativity, and distributivity. Furthermore, FIRA contains additional operators for data-metadata transformations that are more general and widely applicable than predecessor languages [19].

1.1 Main Contributions

Our contributions in this paper are as follows.

- We describe a modular, non-intrusive implementation of a FISQL/FIRA query engine (Figure 10). Our engine supports full dynamic-schema FISQL, including sub-queries and multi-source input.
- We introduce novel, generalized FIRA rules for rewriting dynamic schema queries and subqueries throughout Section 3. We organize these into two optimization *phases* of FIRA plan rewrites for equivalent FISQL.
- We introduce a *bi-level* optimization paradigm for exploiting source relational query optimizers (Section 2.3). This approach shares some features of canonical distributed database query processing [18], however innovations were required for handling dynamic schemas.
- We give experimental results showing the feasibility of extended relational optimization using FIRA. Our experiments show that the generalized algebraic rewrites yielded by FIRA achieve feasible plans and interesting tradeoffs in the context of distributed, dynamic source schemas (see Section 4).

CA
ORDERS

CUSTKEY	DATE	TOTALSALE
SMITH	2005-08-01	955
SMITH	2005-08-02	589
JONES	2005-08-01	938
BROWN	2005-08-01	23
BROWN	2005-08-02	55
BLACK	2005-08-01	999
BLACK	2005-08-02	300
HENRY	2005-08-02	935

IL
ORDERS

DATE	SMITH	HENRY	JAMES	MILLS	BURNS
2005-08-01	967	⊥	689	957	68
2005-08-02	566	953	18	45	⊥

NY

SMITH

DATE	TOTALSALE
2005-08-01	957
2005-08-02	532

BROWN

DATE	TOTALSALE
2005-08-01	230
2005-08-02	551

JAMES

DATE	TOTALSALE
2005-08-01	699
2005-08-02	81

BURNS

DATE	TOTALSALE
2005-08-01	86
2005-08-02	978

MILLS

DATE	TOTALSALE
2005-08-01	866
2005-08-02	595

WEBER

DATE	TOTALSALE
2005-08-01	908
2005-08-02	139

Figure 1: Multidatabase containing information on customer orders.

Our results show that metadata transformations can effectively and efficiently be carried out within the relational model (see Section 4). An important application of our work is to facilitate data integration in frameworks incorporating mappings between source databases. In particular, our work can facilitate GAV, LAV, GLAV [12], BAV [15], as well as peer-to-peer approaches, by non-intrusively extending source RDBMS with dynamic schema capabilities so that each becomes a *unit of interoperability* more robust under schema transformation and evolution. Adding the flexibility and transformation space of FISQL to source engines would give broader support for the mappings that are crucial for integration frameworks. A caveat is that our work focuses on *structural* integration, whereas *semantic* reconciliation remains one of the toughest problems facing data integration [13]. However, researchers have addressed this with some success, and our framework complements systems such as Clio [16] or iMAP [4].

Finally, we give a brief note on terminology. The term *multidatabase* [14] has come to mean a collection of (relational) databases containing semantically similar but possibly structurally dissimilar information. On the other hand, the term *federated* sometimes carries the connotation of a common global schema [17]. In this paper, we use the terms "federation" and "multidatabase" interchangably to mean a collection of databases containing semantically related information, and make no commitment to any overarching global schema or network architecture.

2. FISQL/FIRA

For this paper, we assume an underlying domain of atomic elements, denoted Ω. Elements of Ω are assumed to be alphanumeric strings and will be written in all-caps, as in Figure 1. Our implementation currently supports both string and integer types, and in fact even allows both to occur as column or relation names,[1] or together in a single column. However, this extended typing is orthogonal to FIRA optimization and so is discussed in a separate paper [21]. We assume distinct elements "$\perp$" (the null value), where $\perp \notin \Omega$, and also $\epsilon \in \Omega$ (the empty string).

One of the main features of the federated data model underpinning FISQL and FIRA is that *both data and metadata are taken from* Ω. This seems natural, since metadata is usually thought of as strings, and enables us to formally compare data and metadata within the model.

Both FISQL and FIRA include the ability to create and restructure relational schemas dynamically, at run time. In order to support this, [19] defines a broader notion of *schema* within an extended relational data model known as the *federated data model*. Here, we assume the following definitions.

Definition 2.1

1. A (federated) *tuple*, t, is a mapping $t : \Omega \rightarrow \Omega \cup \{\perp\}$ that maps all but a finite number of elements in its input domain to $\perp$. The *active schema* of t, denoted $asch(t)$, is the non-null mapped input elements.

2. A (federated) *relation*, r, consists of a name (denoted $name(r) \in \Omega$), and a body (denoted $body(r)$). The relation body is a finite set of (federated) tuples. The *active schema* of r, denoted $asch(r)$, is $\cup_{t \in r} asch(t)$.

3. A (federated) *database*, d, is a finite set of (federated) relations.

4. A *federation* is a finite set of (federated) databases.

Note that the "full" schema of every input relation is the whole of Ω. One consequence is the fact that relation schemas always match; hence FIRA versions of $\times$, $\cup$ and $-$ are always "outer" (see Section 2.2.1). The federated data model is robust under dynamic schema creation, which is its main appeal (see [19] for details). By convention, a federated relation is finitely represented as a table containing columns with at least one non-null value.

[1]Note that we allows numbers to be promoted to column headings or relation names. Although this seems strange at first, it is a natural consequence of dynamic schema creation. For example, in spreadsheet data numeric column headings (such as dates or quarters) are heavily used.

2.1 FISQL

A full EBNF for FISQL as we implemented it is given in Appendix A. *Metavariables*, which range over attribute or relation names, are declared using a ":" in the `from` clause. When these attribute or relation names are fixed, the specific value may be substituted in the `from` clause and signified with a "." instead of a ":".[2] A full presentation of FISQL is available in [19]. The following examples summarize the relevant syntactic features.

Example 2.1 (IL → CA)
The following query shows the FISQL syntax for declaring and using an attribute metavariable. This is the transformation from the IL to the CA database.

```
select Iatt as 'CUSTKEY',
       I.DATE as 'DATE',
       I.Iatt as 'TOTALSALE'
 into  'NewIL'
from   IL.ORDERS:Iatt as I
where  Iatt <> 'DATE'
```

Example 2.2 (NY → CA)
The following query shows the FISQL syntax for declaring and using a relation metavariable. This is the transformation from the NY to the CA database.

```
select Nrel as 'CUSTKEY',
       N.DATE as 'DATE',
       N.TOTALSALE as 'TOTALSALE'
 into  'NewNY'
from   NY:Nrel as N
```

Example 2.3 (CA, IL Comparison with Subquery)
The following query finds customers who spent more than $950 on the same date according to both the CA and IL databases. Note the subquery uses the FISQL "`on`" keyword for dynamic attribute creation.[3]

```
select Iatt as 'CUSTKEY'
 into 'RESULT'
from  (select T.DATE as 'DATE',
              T.TOTALSALE on T.CUSTKEY
         into 'NewCA'
        from  CA.ORDERS as T
       ).NewCA:Catt as C,
      IL.ORDERS:Iatt as I
where I.DATE = C.DATE and
      Iatt = Catt
      I.Iatt > 950 and
      C.Catt > 950
```

Of course, this query can be phrased without the subquery. However, a human user might naturally prefer to use a more basic subquery to first perform the intended restructuring and then do the comparison in the outer query, which makes the overall query easier to understand. A strength of our implementation is that even if the input query is inherently inefficient in this way, our optimization techniques will automatically reduce it to a much simpler form (see Section 3, Example 3.1).

Example 2.4 (CA, NY Comparison with Subquery)
Similarly, the following query employs the transformation from the NY database to the CA database to find customers who have spent more than $950 on the same date according to both those databases. Note the inner query dynamically creates a database as the term to the right of the `into` keyword dynamically resolves to the `CUSTKEY` values.

```
select Nrel as 'CUSTKEY'
 into  'RESULT'
from   (select T.DATE, T.TOTALSALE
          into  T.CUSTKEY
         from   CA.ORDERS as T
         ):Crel as C,
        NY:Nrel as N
where  N.DATE = C.DATE and
       Nrel = Crel and
       N.TOTALSALE > 950 and
       C.TOTALSALE > 950
```

2.2 FIRA

For convenience and consistency, in this section we present a brief summary of FIRA and discuss how FISQL input queries are translated to FIRA query plans in our implementation. Note that the semantics of some FIRA operators differs subtly from the presentation in [19]. In particular, for some operators, it was recognized that either a "no-overwrite" or an "overwrite" semantics could be consistent for FIRA. In the implementation, it became necessary to choose one of these semantics. The one that facilitates query optimization best is a *no-overwrite* semantics where applicable. This is detailed in the presentation below.

Also, the presentation in [19] requires a notion of *meta-metadata*. With the *no-overwrite* semantics in place, we can remove this additional complexity and instead use ordinary metadata, as detailed below.

Like FISQL, a FIRA query takes *databases* as input and returns a single *database*. This closure is a natural parallel of SQL/RA closure since SQL and RA take *relations* as input and return a single *relation*.

2.2.1 Relational Core of FIRA

FIRA contains federated versions of the RA operators extended to multidatabases, which we denote using "hats" (following [19]). One of the fundamental properties of the FISQL/FIRA framework is that RA is equivalent to a subalgebra of FIRA (namely, that delineated by the federated RA counterparts). For the purposes of this paper, this means RA algorithms and equivalences carry over directly to FIRA. This enables us to elegantly use and extend the considerable body of research on canonical relational query optimization.

In the following definitions, we will use the notation $\langle n, x \rangle$ where $n \in \Omega$ and x is a set of (federated) tuples to mean the federated relation with name n and body x.

For the unary RA operators, Π and σ, the federated counterparts simply map the relational operations to the bodies of constituent relations.[4]

[2] The "." notation is shorthand for a ":" declaration in the `from` clause followed by an equality constraint in the `where` clause.

[3] To exactly reproduce the format of the IL.ORDERS relation in the subquery, an operation to appropriately merge tuples with null values would be required. Adding a *merge* operation is problematic for several reasons, but can be done successfully [20]. For many useful comparisons a diagonal transpose suffices (as here). Indeed, one of the strengths of the FISQL/FIRA framework is that it breaks PIVOT-type operations into more fundamental components so that these can be uniformly manipulated with algebraic rewrite rules (see Sections 2.2.2 and 5 as well as [20]).

[4] FIRA includes an extended renaming operation, $\hat{\rho}$. For conciseness we do not include this operation here.

Definition 2.2 (Unary Federated RA Operations) Let d be a (federated) database instance, θ be a selection condition, and $X \subseteq \Omega$ a finite set of atoms.

1. (Federated Selection)

$$\hat{\sigma}_\theta(d) = \{\langle name(r), \sigma_\theta(body(r))\rangle | \ r \in d\}.$$

2. (Federated Projection)

$$\hat{\Pi}_X(d) = \{\langle name(r), \Pi_X(body(r))\rangle | \ r \in d\}.$$

Note that for the federated model, the columns in a projection have not "dissapeared" as in the canonical model; rather, the effect of a project is to set all values in non-projected columns to $\perp$.

In the examples below, we will use a convenient shorthand which combines projection and renaming. The notation $\Pi^B_A(r)$ means that r is first projected on A, and then the A column is renamed to B (where $A, B \in \Omega$ and $B \notin asch(r)$).

For the binary RA operators, $\times$, $\cup$, and $-$, non-trivial generalizations are needed. The federated counterparts for these operators only combine relations from the input databases where those relations *have the same name*. This subtle but important divergence from previous multidatabase operators such as MRA [7] preserves important properties such as commutativity of $\bowtie$.

Definition 2.3 (Binary Federated RA Operations) Let d_1 and d_2 be two (federated) database instances.

1. (Federated Cartesian Product)

$$d_1 \hat{\times} d_2 = \{\langle name(r_1), body(r_1) \times body(r_2)\rangle | \ r_1 \in d_1, r_2 \in d_2 \text{ and } name(r_1) = name(r_2)\}.$$

2. (Federated Set Union)

$$\begin{aligned} d_1 \hat{\cup} d_2 = &\{\langle name(r_1), body(r_1) \cup body(r_2)\rangle | \\ &r_1 \in d_1, r_2 \in d_2 \text{ and } name(r_1) = name(r_2)\} \\ &\cup \{r_1 \in d_1 | \ \nexists r_2 \in d_2 . name(r_2) = name(r_1)\} \\ &\cup \{r_2 \in d_2 | \ \nexists r_1 \in d_1 . name(r_1) = name(r_2)\}. \end{aligned}$$

3. (Federated Set Difference)

$$\begin{aligned} d_1 \hat{-} d_2 = &\{\langle name(r_1), body(r_1) - body(r_2)\rangle | \\ &r_1 \in d_1, r_2 \in d_2 \text{ and } name(r_1) = name(r_2)\} \\ &\cup \{r_1 \in d_1 | \ \nexists r_2 \in d_2 . name(r_2) = name(r_1)\}. \end{aligned}$$

As previously mentioned, union or difference operations always succeed on federated relations, since the full schema of these relations is always Ω. Thus, these operations are *outer* in that columns from one relation not appearing in the other are assumed to exist and contain only null values.

For examples and further discussion of the relational core of FIRA, please see [19].

2.2.2 Extended Relational Operators

In addition to the relational core, FIRA contains six new operators for creating and manipulating dynamic schemas.

Metadata Demotion. For a (federated) relation, r, and $A, B \in \Omega$ let $metadata^B_A(r)$ be the relation with schema $\{A, B\}$ containing a tuple of the form $\langle A : name(r), B : a\rangle$ for each $a \in asch(r)$.

Definition 2.4 (Metadata Demotion) Let d be a (federated) database instance and $A, B \in \Omega$. Then $\downarrow^B_A (d)$, the *metadata demotion* over d is:

$$\downarrow^B_A (d) = \{\langle name(r), metadata^B_A(r) \times body(r)\rangle | r \in d\}.$$

No-Overwrite Semantics: In the case where A or B already exists in the relation being paired with its metadata, we assume the operation has *no effect* on that column. In other words, previously existing columns in r are *not* overwritten in a metadata demotion.

In the FIRA query plans for our examples, we will use two distinct versions of the $\downarrow$ operator. $\downarrow_A$ (the subscript version) is assumed to pair only the attribute names with the relation body. $\downarrow^B$ (the superscript version) is assumed to pair only the relation name with its body.

Drop Projection. In addition to ordinary projection, the dynamic nature of federated schemas necessitates adding a *Drop Projection* to FIRA, denoted Π, and defined as follows.

Definition 2.5 (Drop Projection) Let d be a (federated) database and $A \in \Omega$.

$$\Pi_A(d) = \{\langle name(r), \Pi_{asch(r)-\{A\}}(r)| \ r \in d\}.$$

Dynamic Attributes. FIRA contains an operator for adding attributes and associated contents dynamically to a relation (τ). In addition, FIRA includes an operator for *dereferencing* a column of values in a relation, which is effectively a projection that can project a different column for each tuple in a relation (Δ).

Definition 2.6 (Diagonal Transpose)

1. Let r be a (federated) relation and $A, B \in \Omega$. Then the *Diagonal Transpose* of r, denoted $\tau^B_A(r)$, is the set of tuples t' where for each $t \in r$ and for each $C \in \Omega$, t' is constructed as follows.

$$t'[C] = \begin{cases} t[A] \text{ when } C = t[B] \text{ and } C \notin asch(r), \\ t[C] \text{ otherwise.} \end{cases}$$

2. For a (federated) database, d,

$$\tau^B_A(d) = \{\langle name(r), \tau^B_A(r)\rangle | r \in d\}.$$

Note the *no-overwrite* semantics for transposition: if any values being promoted already exist in the active schema, the corresponding columns are left untouched.

Example 2.5
Figure 2 (a) shows the result of a diagonal transpose on the `CA.ORDERS` table.

Our operator τ has affinities with the PIVOT operator available in commercial RDBMS and spreadsheet applications [2]. One of the contributions of FIRA is to decompose PIVOT into more fundamental operations, beginning with a

CUSTKEY	DATE	TOTALSALE	SMITH	JONES	BROWN	BLACK	HENRY
SMITH	2005-08-01	955	955	⊥	⊥	⊥	⊥
SMITH	2005-08-02	589	589	⊥	⊥	⊥	⊥
JONES	2005-08-01	938	⊥	938	⊥	⊥	⊥
BROWN	2005-08-01	23	⊥	⊥	23	⊥	⊥
BROWN	2005-08-02	55	⊥	⊥	55	⊥	⊥
BLACK	2005-08-01	999	⊥	⊥	⊥	999	⊥
BLACK	2005-08-02	300	⊥	⊥	⊥	300	⊥
HENRY	2005-08-02	935	⊥	⊥	⊥	⊥	935

(a)

DATE	SMITH	HENRY	JAMES	MILLS	BURNS	NEW1	NEW2
2005-08-01	967	⊥	689	957	68	DATE	2005-08-01
2005-08-01	967	⊥	689	957	68	SMITH	967
2005-08-01	967	⊥	689	957	68	HENRY	⊥
2005-08-01	967	⊥	689	957	68	JAMES	689
2005-08-01	967	⊥	689	957	68	MILLS	957
2005-08-01	967	⊥	689	957	68	BURNS	68
2005-08-02	566	953	18	45	⊥	DATE	2005-08-02
2005-08-02	566	953	18	45	⊥	SMITH	566
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(b)

Figure 2: (a) $\tau^{\mathbf{DATE}}_{\mathbf{TOTALSALE}}$(CA.ORDERS), (b) $\Delta^{\mathbf{NEW2}}_{\mathbf{NEW1}}(\downarrow_{\mathbf{NEW1}}$ (IL.ORDERS))

diagonal transpose (see [20]). This enables us to fully characterize the behaviour of PIVOT on general input relations, generalize PIVOT to cases without aggregate functions, and determine the complexity of PIVOT in general [20]. In [20], FIRA is extended with an *optimal tuple merge* operator, μ. Future work will add μ to our FIRA engine (see Section 6).

Definition 2.7 (Dereference Projection)

1. Let r be a (federated) relation and $A, B \in \Omega$. Then the *Dereference Projection* of r, denoted $\Delta^B_A(r)$, is the set of tuples t' where for each $t \in r$ and for each $C \in \Omega$, t' is constructed as follows.

$$t'[C] = \begin{cases} t[t[A]] \text{ when } C = B \text{ and } B \notin \mathit{asch}(r), \\ t[C] \text{ otherwise.} \end{cases}$$

2. For a (federated) database, d,

$$\Delta^B_A(d) = \{\langle \mathit{name}(r), \Delta^B_A(r)\rangle | r \in d\}.$$

Once again, note the *no-overwrite* semantics for Δ.

Example 2.6
Figure 2 (b) shows the result of an attribute demotion followed by a dereference projection on `IL.ORDERS`. Note that a dereference projection may result in columns with heterogeneous atomic types. Our implementation supports this, for details see [21].

Including Dereference in Selection and Projection. A convenient shorthand was introduced in [19] for including dereference columns directly in selection conditions and projection lists. This is the "overbar" notation. For example, for $A, B \in \Omega$ and r a (federated) relation instance, the term $\sigma_{\overline{A=B}}(r)$ means we first dereference the column A, and then compare the resulting new column to B. In other words, this term is equivalent to $\sigma_{C=B}(\Delta^C_A(r))$ where $C \in \Omega$ and $C \notin \mathit{asch}(r)$. Similarly, the "overbar" notation may be used in a projection, where $\Pi^B_{\overline{A}}(r)$ means $\Pi^B_C(\Delta^C_A(r))$. Note that in such projection terms the atom B must be included in the superscript, fixing the renaming that occurs.

Not only is the overbar notation convenient for the translation from FISQL to FIRA, but in fact "$\sigma_{\overline{A=B}}(r)$" and "$\Pi^B_{\overline{A}}(r)$" have separate implementation algorithms which are more efficient than a Δ followed by a σ or Π. This is similar to the manner in which a $\bowtie$ operation is more efficient than a $\times$ followed by a σ (see Section 3).

Dynamic Relations. Finally, FIRA contains operations for splitting a single relation into several, and also for unioning several into one, as follows.

Definition 2.8 (Partition a Relation)

1. Let r be a (federated) relation and $A \in \Omega$. The result of partitioning r on A, denoted $\wp_A(r)$, is a database obtained as follows.

$$\wp_A(r) = \{\langle a, \sigma_{A=a}(\mathit{body}(r))\rangle | \exists t \in \mathit{body}(r).t[A] = a\}.$$

2. For a (federated) database, d,

$$\wp_A(d) = \hat{\bigcup}_{r \in d} \wp_A(r).$$

Definition 2.9 (Generalized Union)
Let d be a (federated) database and recall $\epsilon \in \Omega$ is the "empty name". The *generalized union* of d, denoted $\Sigma(d)$, is defined as follows.

$$\Sigma(d) = \{\langle \epsilon, \hat{\cup}_{r \in d} \mathit{body}(r)\rangle\}.$$

That is, $\Sigma(d)$ is a database with a single relation whose name is ϵ and body is the (federated) union of the bodies of every relation in d.

2.2.3 Variable Scoping from FISQL

In the "named" RA, a Cartesian product technically fails when the input relations have overlapping schemas. FIRA inherits this behavior. However, in practise, SQL allows *scoping* of the columns in input relations using the `as` keyword in the `from` clause. This translates to an "entire relation rename" in RA. FISQL similarly allows this type of automatic renaming, or scoping. Our implementation does include a notion of the *scope* of a tuple or metavariable which codifies this idea. However, the range of a FISQL tuple variable is naturally seen as a *database*, not a relation. To remain faithful to this idea in our example query plans, we will subscript Σ operations with a tuple variable, which gives their scope in accordance with the FISQL to FIRA translation.

2.2.4 System-Generated Column Identifiers

In the translation from FISQL to FIRA, intermediate results may have new (temporary) columns as a result of the decomposition into FIRA operators. These are supported in the implementation and employ a separate namespace to avoid collisions. In FIRA query plans (for example Figure 3, below), system-generated identifiers will be prefixed with double underscores (for example "__Iatt") to distinguish them from identifiers appearing in the input query. Note that the appearance of such identifiers is an artifact of the translation from FISQL to FIRA, and does *not* add expressive power to FIRA itself.

2.3 The Federate Interface

One of our goals in processing and optimizing FISQL queries is to take advantage as much as possible of source query engines, which will likely include highly sophisticated query optimizers. This goal forms the basis of our *bi-level* approach to FIRA query optimization. In a FIRA query plan, purely relational operations can be pushed toward the leaves (where source databases come into play). In this case, we can bundle these (purely relational) operations into our request for data from the source. This paradigm is reified in the physical source interface operator Φ.

Definition 2.10 (Federated Source Interface) Let D be a (relational) database schema consisting of relation schemas $R_1, ..., R_n$, and d an instance of D. Let $Q_1, ..., Q_n$ be RA queries over $R_1, ..., R_n$ (respectively). Then we *federate* d using the Φ operator, defined as follows.

$$\Phi(\{Q_1, ..., Q_n\}, d) = \{\langle R_i, Q_i(d)\rangle | 1 \leq i \leq n\}.$$

Here, we overload R_i to mean the *name* of the source relation being returned, as an element of Ω. Thus, Φ applies a filter query to source relations and also pairs them with their name as given by source system tables. Additionally, the filter query may have the form "`select * from *`". The meaning of this is to return all relations in d, with no additional relational filtering.

Filter queries can be combined when the schemas of source relations permit. Note that Φ is in general *not* equivalent to a relational union operation, since result schemas are allowed to vary dynamically. Some care must be taken to respect scope and naming issues when pushing relational query fragments into Φ and combining filter queries, but this can be done successfully (see [21] for details on how our implementation introduces a powerful normal form for dealing with such issues).

Example 2.7 (CA, IL Comparision – Naïve Plan) A naïve FIRA query plan for the FISQL query in Example 2.3 is shown in Figure 3. In fact, this translation is *semi-naïve* in that intermediate projections and/or drop projections have been omitted to make the result easier to visualize. Note that leaves contain Φ operations. In the next section, we will show how most of the complexity of such queries can be absorbed into Φ.

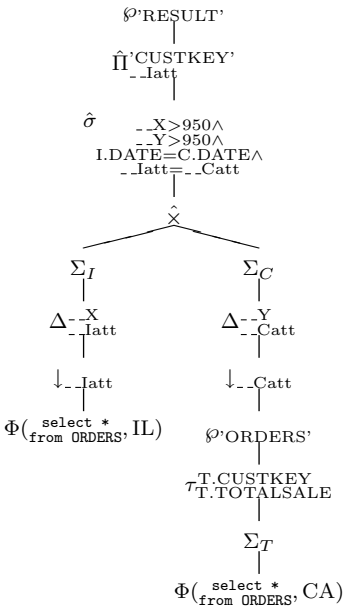


Figure 3: Semi-Naïve FIRA plan for Example 2.3.

Example 2.8 (CA, NY Comparision – Naïve Plan) A semi-naïve FIRA query plan for the the FISQL query in Example 2.4 is shown in Figure 4.

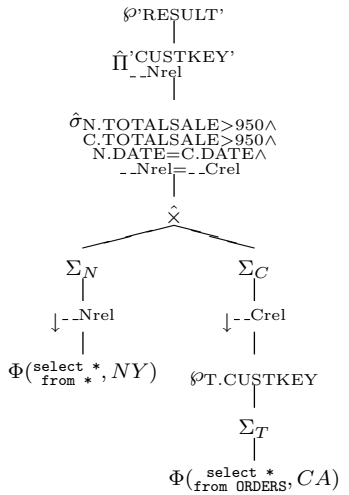


Figure 4: Semi-Naïve FIRA plan for Example 2.4.

3. DYNAMIC QUERY OPTIMIZATION

Our optimizer utilizes a "two-phase" rewriting scheme. The first phase follows the fundamental techniques of canonical relational optimization, but rewrite rules are augmented significantly with support for FIRA operations as well as RA. In phase I, $\hat{\sigma}$, $\hat{\Pi}$, and Π constraints are rewritten to reduce the size of intermediate expressions. Intutively, this phase "pushes" constraining expressions as far down the FIRA query plan tree as possible, including (where possible), into Φ expressions. Furthermore, selection conditions and Cartesian products are combined into $\hat{\bowtie}$ operations. This phase of the optimization is detailed in Section 3.1.

In the second phase (Section 3.2), metavariables are removed from the input query where possible. This can often be achieved through syntactic rewrites arising from equality σ conditions, which is a *write-out* strategy. Alternatively (or in combination), metavariables over static schemas may be *unwound* based on the source system tables.

All optimizations are based on a cost model involving both the number of tuples in the input relations as well as the number of columns. Many FIRA operations have analyses directly paralleling their RA counterparts, except that a new factor of the width of the relation is involved. Detailed algorithms and analyses of these are deferred to another paper [21], since the focus of the current paper is on the high-level algebraic rules employed during optimization. Practically speaking, one result of our analyses is that the $\downarrow$ operation is the most expensive (after $\hat{\times}$). Thus, one goal is to eliminate and/or minimize metadata demotions wherever possible.

All proofs of the rewrite rules that follow are deferred to a longer version of the paper. Furthermore, for brevity, all rewrite rules involving operators purely from the RA subalgebra of FIRA are omitted. Note, however, that all such rules have non-trivial generalizations for including "overbar" terms in selection and projection arguments.

3.1 Phase I Rewrite Rules

3.1.1 Selection Rewrite Rules

In what follows, we use the notation θ to refer to a generic selection condition. Furthermore, the notation $\theta|_A^B$ means the result of syntactically replacing all occurrences of A in θ by B. For this substitution to be valid, it must be the case that B is not already in θ, of course. If A does not occur in θ, the substitution has no effect.

Proposition 3.1 (Basic Rewrites)
Let $A, B \in \Omega$ and θ be a selection condition not involving A or B. Let d be a (federated) database instance. Then:

$$\hat{\sigma}_\theta(\wp_A(d)) = \wp_A(\hat{\sigma}_\theta(d)) \quad (1)$$

$$\hat{\sigma}_\theta(\Sigma(d)) = \Sigma(\hat{\sigma}_\theta(d)) \quad (2)$$

$$\hat{\sigma}_\theta(\tau_A^B(d)) = \tau_A^B(\hat{\sigma}_\theta(d)) \quad (3)$$

$$\hat{\sigma}_\theta(\Delta_A^B(d)) = \Delta_A^B(\hat{\sigma}_\theta(d)) \quad (4)$$

$$\hat{\sigma}_\theta(\Pi_A(d)) = \Pi_A(\hat{\sigma}_\theta(d)) \quad (5)$$

$$\hat{\sigma}_\theta(\downarrow_A^B (d)) = \downarrow_A^B (\hat{\sigma}_\theta(d)) \quad (6)$$

The following rules hold because the pair of operations Δ and $\downarrow$ are essentially inverse to τ (see [20]).

Proposition 3.2 (Dynamic Attribute Specialization)

1. Let $A, B, C, D \in \Omega$, θ be a selection condition not involving A, B, or C, and d be a (federated) database instance. Then:

$$\hat{\sigma}_\theta(\Delta_C^D(\downarrow_C (\tau_A^B(d)))) = \Delta_C^D(\downarrow_C (\tau_A^B(\hat{\sigma}_{\theta|_D^A}(d)))). \quad (7)$$

2. Let $A, B, C \in \Omega$, θ be a selection condition not involving A or B, and d be a (federated) database instance.

$$\hat{\sigma}_\theta(\downarrow_C (\tau_A^B(d))) = \downarrow_C (\tau_A^B(\hat{\sigma}_{\theta|_C^B}(d))). \quad (8)$$

Example 3.1
Figure 5 shows the result of pushing selections down the tree from Figure 3 according to the above rules. Note that rule (7) enabled us to push the "> 950" condition on the dereference all the way into the Φ operation on the CA database.

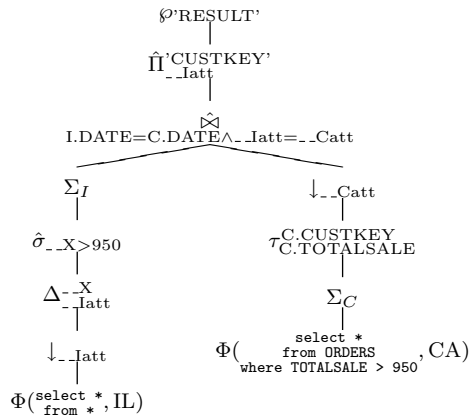


Figure 5: Fully constrained plan for Example 2.3.

Proposition 3.3 (Dynamic Relation Specialization)
Let $A, B \in \Omega$ and θ be a selection condition that does not involve A. Let d be a (federated) database instance. Then:

$$\hat{\sigma}_\theta(\downarrow^B (\wp_A(d))) = \downarrow^B (\wp_A(\hat{\sigma}_{\theta|_B^A}(d))). \quad (9)$$

Example 3.2
Figure 6 shows the result of pushing selections down the tree from Figure 4 according to the above rules. Note that rule (9) enabled us to push the "> 950" condition on the dereference all the way into the Φ operation on the CA database.

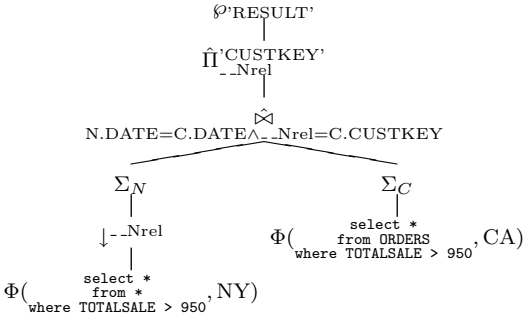


Figure 6: Fully constrained plan for Example 2.4.

3.1.2 Projection Rewrite and Removal Rules

Proposition 3.4
Let $X \subseteq \Omega$ be a finite set of atoms and $A, B \in \Omega$ where $A, B \notin X$. Let d be a (federated) database instance. Then:

$$\hat{\Pi}_X(\wp_A(d)) = \wp_A(\hat{\Pi}_X(d)) \tag{10}$$

$$\hat{\Pi}_X(\Sigma(d)) = \Sigma(\hat{\Pi}_X(d)) \tag{11}$$

$$\hat{\Pi}_X(\tau_A^B(d)) = \tau_A^B(\hat{\Pi}_X(d)) \tag{12}$$

$$\hat{\Pi}_X(\Delta_A^B(d)) = \Delta_A^B(\hat{\Pi}_X(d)) \tag{13}$$

$$\hat{\Pi}_X(\Pi_A(d)) = \Pi_A(\hat{\Pi}_X(d)) \tag{14}$$

$$\hat{\Pi}_X(\downarrow_A^B(d)) = \downarrow_A^B(\hat{\Pi}_X(d)) \tag{15}$$

Each of the rules in Proposition 3.4 has a counterpart for Π. For lack of space, these are not given here.

Finally, through successive rewriting some operations will become superfluous. The following rules determine when such operations can simply be removed from a query plan. Note that these rules must be applied on a per-relation basis, as they involve statically determining the active schema of that relation. In our examples, this is straightforward, as static type inference in FIRA is well behaved in many cases. For details, please refer to the longer version of this paper.

Proposition 3.5
Let r be a (federated) relation, $X \subseteq asch(r)$, and $A, B \in \Omega$.

$$\hat{\Pi}_X(\tau_A^B(r)) = \hat{\Pi}_X(r) \tag{16}$$

$$\hat{\Pi}_X(\Delta_A^B(r)) = \hat{\Pi}_X(r) \text{ for } B \notin X \tag{17}$$

$$\hat{\Pi}_X(\downarrow_A^B(r)) = \hat{\Pi}_X(r) \text{ for } A, B \notin X \tag{18}$$

$$\Pi_B(\Delta_A^B(r)) = r \tag{19}$$

$$\Pi_{A,B}(\downarrow_A^B(r)) = r \tag{20}$$

Finally, since Σ and $\wp$ are essentially inverses, we can remove redundant $\wp$ operations as follows.

Proposition 3.6
Let d be a (federated) database and $A \in \Omega$. Then:

$$\Sigma(\wp_A(d)) = \Sigma(d) \tag{21}$$

3.2 Phase II Rewrite Rules

3.2.1 Metavariable "Write-Out" Rule

Proposition 3.7 (Metavariable Write-Out)
Let θ be a selection condition and $A, B, C \in \Omega$. Let d be a (federated) database instance. Then:

$$\hat{\sigma}_{A=C \wedge \theta}(\Delta_A^B(\downarrow_A(d))) = \hat{\sigma}_{(\theta|_A^C)|_B^{\overline{C}}}(d). \tag{22}$$

The main use of this rule is to remove the expensive $\downarrow$ operation where possible. The selection-with-overbar operation has a separate, efficient implementation as a single, composite operation. Thus, the rule in Proposition 3.7 enables us to employ this composite operator. The overall impact is similar to rewriting a $\times$ and σ pair of operations in the original RA as a $\bowtie$. Semantically, the operations are the same, but $\bowtie$ has a separate implementation that is more efficient than a $\times$ operation followed by a σ. Similarly, a $\hat{\sigma}_{\overline{*}}$ is more efficient than a $\downarrow$ followed by a Δ followed by a $\hat{\sigma}$.

Example 3.3
Consider the following query, which finds customers that have spent more than $950 on two different dates according to both the CA and IL databases.

```
select T1.CUSTKEY as 'CUSTKEY' into 'RESULT'
from   CA.ORDERS as T1, CA.ORDERS as T2,
       IL.ORDERS:A3 as T3, IL.ORDERS as T4
where  T1.CUSTKEY = T2.CUSTKEY and T1.TOTALSALE > 950 and
       T2.TOTALSALE > 950 and T1.DATE <> T2.DATE and
       A3 = T1.CUSTKEY and A3 <> 'DATE' and T3.A3 > 950
       and T4.A3 > 950 and T3.DATE <> T4.DATE
```

Using rules (1) through (21), we can obtain the fully constrained query plan shown in Figure 7, where the filter query Q issued to database CA is:

```
select T1.CUSTKEY
from   ORDERS as T1, ORDERS as T2
where  T1.CUSTKEY = T2.CUSTKEY and
       T1.DATE <> T2.DATE and
       T1.TOTALSALE > 950 and
       T2.TOTALSALE > 950
```

However, we can go beyond this and apply rule (22) to obtain the better plan shown in Figure 8.

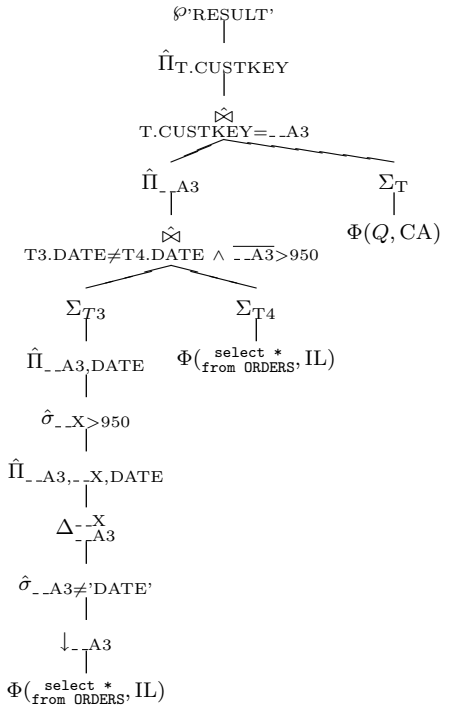


Figure 7: Fully constrained plan for Example 3.3

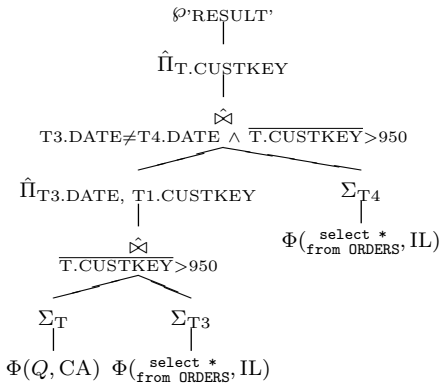


Figure 8: Metavariable-free plan for Example 3.3

3.2.2 *Metavariable "Unwind" Rule*

Proposition 3.7 can only be applied when there is a matching equality constraint on an attribute metavariable, thus limiting its applicability. A more widely applicable technique for metavariable removal is that of *unwinding*. In the case where metavariables range over static metadata, the following *unwind rule* may be applied.

Proposition 3.8 (Metavariable Unwinding)
Let $A, B \in \Omega$ and d be a (federated) database instance. Then:

$$\downarrow_A^B (d) = \hat{\bigcup_{r \in d}} \hat{\bigcup_{a \in asch(d)}} \{\langle A : a,\ B : name(r)\rangle\} \times r. \quad (23)$$

In Proposition 3.8, the notation "$\{\langle A : a,\ B : name(r)\rangle\}$" means the constant relation with schema (A, B) that contains a single tuple $\langle a, name(r)\rangle$. The main advantage of unwinding is that metavariables that can be unwound result in expressions that can be wholly pushed into Φ, fully taking advantage of source query optimizers.

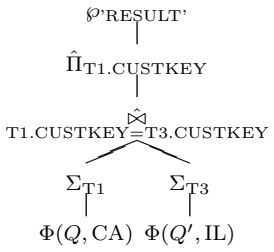
Example 3.4
Figure 9 shows the result of unwinding the query in Example 3.3, as an alternative to the plan of Figure 8.

When both the write-out and unwind techniques apply, there is the question of which to apply (or whether to employ a combination of both techniques). The write-out technique seems more natural at first glance, and is robust as the number of attributes and/or relations in the source catalog grows. In contrast, metavariable unwinding can result in very large union queries. We study the trade-offs involved in the approaches in Section 4.2.1. and present techniques for dealing with large union queries in Φ in Section 4.2.2.

4. EMPIRICAL RESULTS

4.1 System Architecture

Figure 10 depicts the architecture of our FISQL/FIRA engine. We currently have an implementation written in C# using the Microsoft .NET framework [22]. In terms of the figure, module (1) has been completed using ANTLR [23] to assist with parsing and translation. Modules (3) and (4) have also been fully implemented at this time. Module (2) is



Query Q' issued to database IL is:

```
select 'SMITH' as 'CUSTKEY'
from   ORDERS as O1, ORDERS as O2
where  O1.SMITH > 950 and
       O2.SMITH > 950 and
       O1.DATE <> O2.DATE
union
...
union
select 'BURNS' as 'CUSTKEY'
from   ORDERS as O1, ORDERS as O2
where  O1.BURNS > 950 and
       O2.BURNS > 950 and
       O1.DATE <> O2.DATE
```

Figure 9: Unwound plan for Example 3.3

currently in development. Module (5) supports connections to ODBC sources. The system employs SQLServer 2005 [24] as our workspace database. Note that our FIRA engine performs intermediate processing and storage in the workspace database (as opposed to main memory), so that our engine is fully scalable.

In principle, there is no reason why source databases need be *relational*. Our architecture supports future implementations involving heterogeneous sources (see Section 6).

4.2 Experiments

To evaluate the impact of our optimization heuristics, tests were run on data similar to that in Figure 1. Note that the ORDERS federation is inspired by TPC-H benchmark data. However, TPC-H specifies fixed-schema relations, and for our tests it was important to generate input databases with varying schemas. Metrics for generating the data were as follows.

- The number of customers varied from 25 to 1000.[5] Customer names were chosen from recent census data.
- Values for TOTALSALE were randomly distributed in the range 0 to 1000. Additionally, a random sample of 10% of customers were assumed to buy something on any given date. If a customer did not make a purchase on a date, the TOTALSALE is $\perp$.
- Data was generated for 180 consecutive dates. The range is the same for all databases.

Note that each of the two databases involved in the query were generated independently, and re-created for each test to thwart query plan caching by source engines, which can

[5]Note that the upper limit on the number of customers comes from SQLServer, which supports a maximum of 1024 columns in the IL.ORDERS relation.

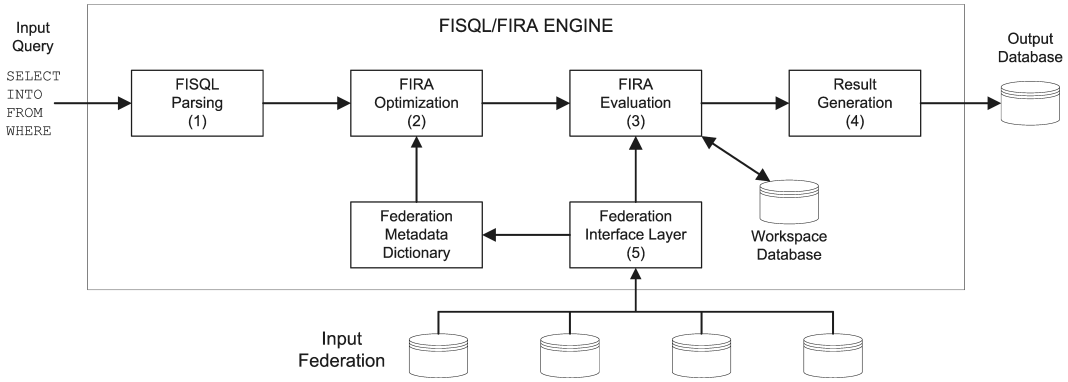


Figure 10: FISQL Processor Architecture

reduce times for the Φ operation by an order of magnitude or more (and so would skew our results).

Tests were run on a system with a 2.8 GHz Pentium D processor and 4 GB RAM running Windows XP (SP2). All databases are hosted on a local SQL Server 2005 (Developer Edition) [24]. Our implementation is in C# using .NET Framework 1.1 [22].

4.2.1 Relative Optimization Performance

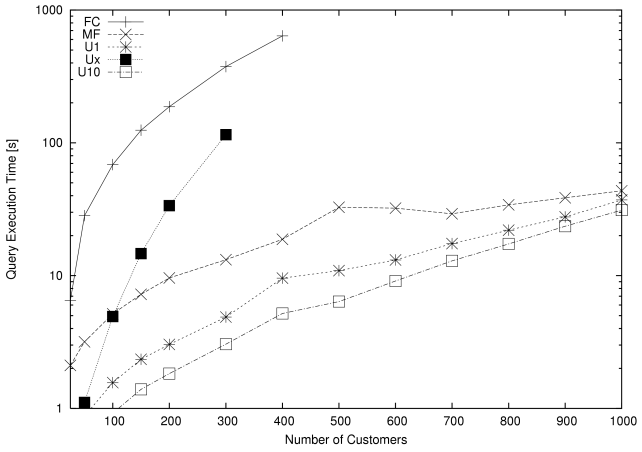


Figure 11: Relative impact of FIRA optimizations on query from Example 3.3. FC = fully constrained plan (Figure 7). MF = metavariable-free plan (Figure 8). Others are unwound plans of varying union batch size (see text for details).

Figure 11 shows query processing times for different FIRA plans for the FISQL query from Example 3.3. Times are shown in seconds for the fully constrained plan from Figure 7 (FC), the metavariable-free plan from Figure 8 (MF), and three versions of the unwound plan from Figure 9 (U1, Ux, U10). The unwound plans differ in the number of unions in the filter query sent by the Φ operator to source database IL as one *batch*. In U1, a single unwound instantiation is sent per batch. In Ux, the filter query is the union of *all* unwound instantiations (so there is a single batch).[6] In U10, each filter query contains the union of 10 unwound instantiations (a value we have found to be roughly optimal for this set-up). This is discussed further below (Section 4.2.2).

Note that the query in Example 3.3 is complex enough to showcase our optimizations, so we have included results for just this query even though our implementation accepts *any* FISQL query. Furthermore, our engine was not optimized for runtime, as our focus was on relative impact of the optimizations.

The fully constrained plan (which still includes FIRA ↓ operations) is considerably slower than the ↓-free plans (processing times of greater than 1000 seconds are not included in the figure). Interestingly, the metavariable-free plan (which still does non-trivial processing in our FIRA engine) is within an order of magnitude of the best unwound plans. This means that, when applicable, dynamic metavariable removal is a real option for FISQL query processing. The advantage of plans that are *not* unwound is that they are independent of the metadata in source databases. For example, plan caching of unwound plans will be be complicated by changes to source metadata (whereas metavariable-free or fully constrained plans will be truly schema independent in this sense).

4.2.2 Effect of Batching on Filter Queries

Figure 12 shows how the overall runtime of Φ varies with the size of filter query batches. Input databases were as in the previous results, fixed for 1000 customers. Along the x-axis, a batch size of "N" means that each filter query contained N unions (arising from metavariable instantiations). The results of these filter queries are combined within Φ itself (in our workspace database) to produce the result. Overall times are shown for the Φ operation on the IL database according to a fully unwound plan for the query in Example 2.3 (IL), as well as for the Φ operation on the NY database according to a fully unwound plan for the query in Example 2.4. Processing times for other FISQL queries showed similar trends according to filter query batch size.

Since we ran tests where sources resided locally, network latency was not an issue. In general, network latency in a distributed setting would need to be taken into consideration when determining the optimal batch size for filter queries (see Section 6).

Interestingly, SQLServer performs well on the *fixed schema*

[6] On batches with more than 300 unions, SQLServer reported that there was insufficient system memory to run the query.

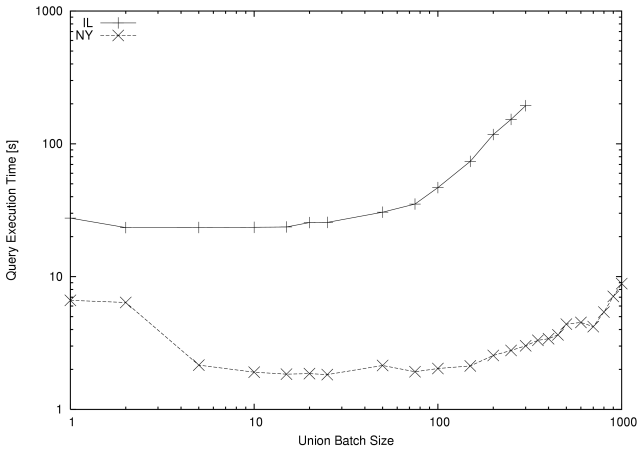


Figure 12: Effect of union batch size on unwound versions of queries from Examples 2.3 and 2.4.

unions arising from the NY database, even when the overall number of unions in the filter query grows. In contrast, for the *dynamic schema* case of the IL.ORDERS table, the SQLServer query engine runs out of memory on filter queries involving more than 300 unions. This highlights one reason why our work is important: canonical RDBMS simply do not deal well with the wide relations that can result from dynamic schema transformations. In contrast, our implementation handles dynamic schemas of arbitrary width by employing an underpinning normal form particularly suited for the federated relational data model (see [21] for details on this normal form).

5. RELATED WORK

To date, very little formalism has been developed to handle dynamic schemas in SQL and RA. FISQL/FIRA is the first set of languages to give a transformationally complete, fully compositional semantics to dynamic relational schemas [19]. The tabular algebra has done this for an extended data model (the tabular data model) [8]. Although not currently implemented (as far as we know), the algebra is important because it is capable of elegantly simulating *all* spreadsheet transformations [8]. Ongoing work is investigating the relationship between FIRA and the tabular algebra.

An important predecessor of FISQL is SchemaSQL [10, 11]. FISQL diverges from SchemaSQL in both syntax and semantics in important ways [19]. As an example, the FISQL `from` clause is *not* like the SchemaSQL metavariable declarations, in that SchemaSQL allows complex interdependent declarations whereas FISQL declarations are independent of one another. The interdependency of SchemaSQL declarations significantly complicates its semantics and is unintuitive for SQL users, as declarations in SQL are independent. Furthermore, SchemaSQL column variables are superfluous and not included in FISQL.

Another crucial difference is that FISQL uses `on` and/or `into` keywords to signal dynamic schemas. In contrast, SchemaSQL uses an extension of the "`create view`" construct for this. In SchemaSQL, a query in the body of a `create view` may change semantics [11]. This means SchemaSQL queries are harder to understand and formalize, and dynamic schemas may not appear in subqueries. Furthermore, SchemaSQL relies on a non-deterministic *merge* operation [11]. Although special cases of *merge* can be deterministic [3, 9, 20], in general *merge* is expensive [20].

Semantically, one of the crucial differences between FISQL and SchemaSQL is that FISQL has an equivalent extended relational algebra, FIRA [19]. Attempts have been made to underpin fragments of SchemaSQL with relational versions of tabular algebra operators [3, 10], such as *fold* and *unfold* [8]. A major limitation of this approach is that the original tabular algebra utilizes a richer data model, so semantic information and complex notions of validity come into play in the SchemaSQL transformation [3, 9, 10, 19]. For example, a *fold* can only be applied to the result of a previous *unfold*, and the columns involved in the *unfold* must have been stored as extra (semantic) information. Nonetheless, some optimization rules can be stated using the restricted tabular algebra operators. These can readily be seen to be special cases of some of the rules developed in Section 3 for FIRA.

These differences explain why reported results of implementations of SchemaSQL are limited to fixed schema, single relation cases, or else procedural simulations of a single, limited, flat dynamic schema operation [10]. In contrast, we have developed and reported results for a fully multidatabase, dynamic schema implementation of FISQL including support for fully compositional, arbitrarily nested subqueries.

There is some work on procedurally unwinding SchemaSQL queries as an implementation strategy [1]. However, the implementation in [1] only investigated the possibility of a single resulting `union` query, and is thus restricted to a special case of our federate operator, Φ.

Our work is related to recent implementations of PIVOT and UNPIVOT [2, 20]. The implementation of dynamic schemas in [2] includes aggregation for collapsing conflicting values in a τ followed by a *merge*. Currently, our implementation of FISQL does not include aggregation, however ongoing work is extending the SQL `group by` operation for inclusion in FISQL. As a beginning, PIVOT and UNPIVOT can be generalized in an extended version of FIRA including a deterministic optimal tuple merge operation [20].

Recent work applies FIRA transformations to delineate a search space for semi-automatically discovering integration mappings [5, 6]. Our engine can be utilized to execute discovered mappings, since these take the form of FIRA queries.

6. CONCLUSION AND FUTURE WORK

In this paper, we have reported on an implementation of FISQL/FIRA for transformationally complete querying in multidatabase systems. Our implementation includes a full FISQL parser and translator, as well as a FIRA evaluation engine, and supports heuristic rule-based optimization in the style of RA optimization. We support dynamic schemas, null values, and heterogeneous atomic types. Optimization methods such as automatic syntactic removal of metavariables, generalized algebraic FIRA plan transformations, and the employment of source filter queries clearly distinguish us from previous, ad hoc approaches involving only metavariable unwinding. Our experiments show that the approach is feasible for TPC type data having dynamic schemas and residing in multiple sources. We are the first to implement a relational query processor and optimizer for dynamic schemas on this scale.

Ongoing work is extending FISQL with aggregation and an optimal tuple merge operator. We plan to include these capabilities in our execution engine. Further interesting projects include studying the impact of network latency on FISQL query processing, as well as augmenting the engine with support for heterogeneous data models, such as semi-structured or nested. In general, the semantics provided by FIRA yields a solid basis for future expansion of our engine.

7. REFERENCES

[1] Francois Barbancon and Daniel P. Miranker. *Implementing Federated Database Systems by Compiling SchemaSQL*. IDEAS 2002.

[2] Conor Cunningham, César A. Galindo-Legaria, and Goetz Graefe. *PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS*. VLDB 2004.

[3] Keir B. Davis and Fereidoon Sadri. *Optimization of SchemaSQL Queries*. IDEAS 2001.

[4] R. Dhamanka, Y. Lee, A. Doan, A. Halevy, and P. Domingos. *iMAP: Discovering Complex Semantic Matches Between Database Schema*. SIGMOD 2004.

[5] George H. L. Fletcher and Catharine M. Wyss. *Relational Data Mapping in MIQIS*. SIGMOD 2005.

[6] George H. L. Fletcher and Catharine M. Wyss. *Data Mapping as Search*. EDBT 2006.

[7] John Grant, Witold Litwin, Nick Roussopoulos, and Timos Sellis. *Query Languages for Relational Multidatabases*. VLDB Journal, vol2. pp 153-171, 1993

[8] Marc Gyssens, Laks V.S. Lakshmanan, and Iyer N. Subramanian *Tables as a Paradigm for Querying and Restructuring*. PODS96, pp. 93-103.

[9] Qi He and Tok Wang Ling. *Extending and Inferring Functional Dependencies in Schema Transformation*. CIKM 2005.

[10] L.V.S. Lakshmanan, F. Sadri, and S.N. Subramanian. *On Efficiently Implementing SchemaSQL on an SQL Database System*. VLDB 1999.

[11] L.V.S. Lakshmanan, F. Sadri, and S.N. Subramanian. *SchemaSQL – an extension to SQL for multidatabase interoperability*. TODS Vol. 26, No. 4, December 2001.

[12] Maurizio Lenzerini. *Data Integration: A Theoretical Perspective*. PODS 2002: 233-246.

[13] Lin Liu. *SIGMOD Record: Special Issue on Semantic Integration*. 33:4, December 2004.

[14] Witold Litwin, M. Ketabchi, and Ravi Krishnamurthy. *First Order Normal Form for Relational Databases and Multidatabases*. SIGMOD RECORD, Vol. 20, No. 4, December 1991.

[15] Peter McBrien and Alexandra Poulovassilis. *Data Integration by Bi-Directional Schema Transformation Rules*. ICDE 2003.

[16] Renée J. Miller, Mauricio A. Hernández, Laura M. Haas, Lingling Yan, C.T. Howard Ho, Ronald Fagin, and Lucian Popa. *The Clio Project: Managing Heterogeneity*. SIGMOD Record, 30:1, March 2001.

[17] Amit P. Sheth and James A. Larson. *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases*. ACM Computing Surveys, 22:3, 1990.

[18] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems, 2nd ed.* Prentice Hall, 1999.

[19] Catharine M. Wyss and Edward L. Robertson. *Relational Languages for Metadata Integration*. TODS, Vol. 30, No. 2, June 2005.

[20] Catharine M. Wyss and Edward L. Robertson. *A formal characterization of PIVOT/UNPIVOT*. CIKM 2005: 602-608.

[21] Catharine M. Wyss and Felix I. Wyss. *Tuple Normal Form for Dynamic Schemas and Heterogeneous Typing in Relational Databases*. Forthcoming.

[22] `http://www.microsoft.com/net`, `http://www.microsoft.com/vcsharp`

[23] `http://www.antlr.org`

[24] `http://www.microsoft.com/sql`

APPENDIX

A. EBNF FOR FISQL

⟨*query*⟩	::=	⟨*query_term*⟩ [(`UNION` | `MINUS`) ⟨*query*⟩]
⟨*query_term*⟩	::=	⟨*query_specifier*⟩
	|	`(` ⟨*query*⟩ `)`
⟨*query_specifier*⟩	::=	`SELECT` [`DISTINCT`] ⟨*column_list*⟩ `INTO` ⟨*column_expr*⟩ [`FROM` ⟨*variable_decl*⟩ {`,`⟨*variable_decl*⟩}* [`WHERE` ⟨*condition*⟩]]
⟨*column_list*⟩	::=	⟨*column_decl*⟩ {`,`⟨*column_decl*⟩}*
	|	`*` [`DROP` ⟨*drop_colref*⟩ {`,`⟨*drop_colref*⟩}*]
⟨*drop_colref*⟩	::=	⟨*identifier*⟩ [`.`⟨*identifier*⟩]
⟨*column_decl*⟩	::=	⟨*column_expr*⟩ [`AS` ⟨*literal*⟩]
	|	⟨*column_expr*⟩ `ON` ⟨*column_expr*⟩
⟨*variable_decl*⟩	::=	⟨*source_db*⟩ [⟨*meta_decl*⟩ [⟨*meta_decl*⟩]] [`AS`] ⟨*identifier*⟩
⟨*source_db*⟩	::=	[`:`]⟨*identifier*⟩
	|	`(` ⟨*query*⟩ `)`
⟨*meta_decl*⟩	::=	`:`⟨*identifier*⟩
	|	`.`⟨*identifier*⟩
⟨*column_expr*⟩	::=	⟨*literal*⟩
	|	⟨*identifier*⟩ [`.`⟨*identifier*⟩ [`.`⟨*identifier*⟩]]
⟨*condition*⟩	::=	⟨*and_cond*⟩ [`OR` ⟨*condition*⟩]
⟨*and_cond*⟩	::=	⟨*unary_cond*⟩ [`AND` ⟨*and_cond*⟩]
⟨*unary_cond*⟩	::=	⟨*comp_expr*⟩
	|	`NOT` ⟨*unary_cond*⟩
	|	`(` ⟨*condition*⟩ `)`
⟨*comp_expr*⟩	::=	⟨*column_expr*⟩ ⟨*comp_op*⟩ ⟨*column_expr*⟩
⟨*comp_op*⟩	::=	`=` | `!=` | `<>` | `<=` | `<` | `>` | `>=`
⟨*literal*⟩	::=	⟨*integer_literal*⟩
	|	⟨*string_literal*⟩
⟨*identifier*⟩	::=	⟨*regular_ident*⟩
	|	⟨*delimited_ident*⟩
⟨*integer_literal*⟩	::=	[`-`|`+`] {⟨*digit*⟩}$^+$
⟨*string_literal*⟩	::=	`'` {∼`'`}* `'` {`'` {∼`'`}* `'` }*
⟨*regular_ident*⟩	::=	(⟨*letter*⟩ |`_`|`@`|`#`) {⟨*letter*⟩ |⟨*digit*⟩ |`_`|`@`|`#`|`$`}*
⟨*delimited_ident*⟩	::=	`"` {∼`"`}* `"` {`"` {∼`"`}* `"` }*
	|	`[` {∼`]`}* `]`
⟨*letter*⟩	::=	`a...z` | `A...Z`
⟨*digit*⟩	::=	`0...9`

Figure 13: EBNF grammar for FISQL as implemented.