5

Extending a Multidatabase Manipulation Language to Resolve Schema and Data Conflicts

Paolo Missier & Marek Rusinkiewicz
Department of Computer Science
University of Houston
Houston, TX 77204-3475
{paolo,marek}@cs.uh.edu

Abstract

The management of Multidatabase Systems (MDBS) is complicated by possible structural and semantic heterogeneity of the member database systems, and the requirements to preserve their local autonomy. Semantic heterogeneity is concerned with the differences in the meaning and interpretation of similar data objects across the different systems. In loosely coupled database federations, static schema integration may be not feasible and heterogeneity conflicts must be dealt with at the application level. Therefore, a multidatabase access language must be equipped accordingly with special features to resolve structural and semantic discrepancies.

In this paper, a multidatabase manipulation language, motivated by Multidatabase SQL, is proposed as a tool for multidatabase access in the presence of schema and data conflicts. Specific features discussed in this proposal include access to external functions, user-defined partial schema integration by means of class attributes, generalized local and interdatabase joins, and automatic completion of local implicit joins. Together, these features permit resolution of many classes of structural schema conflicts that may exist among the member database systems.

Keywords

Heterogeneous Multidatabases, Multidatabase Languages, Schema Conflicts Resolution, Schema Integration.

1 INTRODUCTION

Over the recent years, the growing interest in Multidatabase Management Systems (M-DBMS) and database interoperability motivated a significant amount of research into the issues of database autonomy and schema heterogeneity – DeMichiel (1989), Lim (1993), Ahmed (1991), Krishnamurthy, Litwin and Kent (1991).

As Heimbigner and McLeod (1985) point out, autonomy requirements impose limits on

R. Meersman et al. (eds.), *Database Applications Semantics*
© IFIP International Federation for Information Processing 1997

the degree of information sharing that can be achieved in a multidatabase federation. Heterogeneity pertains both to the technological and structural differences among Database management Systems (DBMS), and to the differences in the semantics of data. Structural heterogeneity results from independent database schema design and the use of different data models for the local schemas. Semantic heterogeneity, as stated by Sheth and Larson (1990), refers to "a disagreement about the meaning, interpretation or intended use of semantically equivalent or semantically compatible data/objects", as well as to data inconsistencies across different schemas and incomplete information.

The two approaches to the reconciliation of the apparently contradictory requirements of a uniform access on one side, and local autonomy and overall heterogeneity of the federation on the other, lead to two different architectural solutions of tightly coupled and loosely coupled database federations. According to Batini, Lenzerini and Navathe (1986), in the former the component (local) schemas are synthesized into a common global schema through a process of schema integration, and the access to component databases is mediated through the global schema. In the latter, no attempt to integrate the local schemas is made, so the coordination of all multidatabase activities takes place at the application level. It is mostly the user's responsibility to access the local systems consistently, through a multidatabase language. This approach seems more general, and hence applicable also in those cases when schema integration does not look feasible, as shown by Litwin and Abdellatif (1986). However, the original proposal by Litwin (1990) for Multidatabase SQL, lack some of the features needed to bridge the heterogeneity gap.

As pointed out by Heiler at el. (1991), the two approaches can be characterized by their respective degree of schema integration transparency. In tightly coupled systems, integration is hidden. The query language emulates a homogeneous access to the federation, and the mappings defined in the global schema assume the burden of reconciliation of discrepancies between the different data models and representations. Conversely, in loosely coupled systems integration is visible, and the access language should be equipped with specific features, adding enough expressivity to deal with application-dependent requirements.

1.1 Assumptions and Goals

In this paper we address the problems of a multidatabase manipulation language with the features necessary to resolve schema and data incompatibilities. Our approach, resulting in extensions to Multidatabase SQL, can be viewed as an intermediate solution between the hidden and visible integration.

In our framework, we do not assume the existence of a Knowledge Base to support the semantic conflict resolution process. In fact, one of the objectives of this work is to investigate the limits of a purely structural approach, in which the DBMS access lànguage provides syntactic mechanisms - through declarations - to establish a correspondence between related objects in the databases of interest to the user.

We will try to characterize those limitations in terms of the kind of database operations supported and the type of conflicts that can be resolved. Although this approach is less comprehensive than the general concept of "integration assisted by mediators" proposed by Wiederhold (1991), it seems to have advantages in its straightforward implementation and ease of use.

In our study, we have focused mainly on read-only queries that reference user-specified,

as well as the standard built-in, relational operators. In order for this technique to scale up, both in the number of integrable schemas and in their complexity, we assume the existence of a "global" Database Administrator (GDBA), or of some other central authority. in the sense indicated by Missier, Rusinkiewicz and Silberschatz (1995). The definition of a single, global schema may be mandated by a "Federation Authority", and the mapping functions from the global to the local schemas are defined locally and independently by the individual DBAs. Those functions can then be referenced by users at query formulation time. In this scenario, MDBMS users are offered a selection of pre-formatted yet customizable views that reflect several ways in which the local and global schemas can be integrated.

The paper is organized as follows. In the next section, we briefly review schema and data incompatibilities and describe existing approaches to heterogeneity resolution that are relevant to our work. We then introduce proposed multidatabase language features (section 3) and describe the evaluation strategy for extended multiqueries (section 3.1). In the last section, we discuss what kinds of heterogeneity can be resolved using the mechanisms we have introduced and outline some possible directions for further work.

2 RELATED WORK IN STRUCTURAL AND SEMANTIC HETEROGENEITY

A classification of the most common heterogeneity conflicts is a good starting point for understanding the expressive power required in a multidatabase access language. A comprehensive classification of schema heterogeneities has been proposed by Kim and Seo (1991). Sheth and Kashyap (1992) introduce the concept of *semantic proximity* that can serve as a measure of distance between entities in different schemas and apply it to analyze the relationship between semantic and structural heterogeneity. These conflicts can be broadly divided into the two classes of *Domain definition* and *Entity definition* incompatibility.

Domain definition conflicts (also referred to as attribute definition conflicts) include naming (synonyms and homonyms), data type, data scaling (using different units of measure), data precision (granularity), default value and attribute integrity constraint problems.

Entity definition conflicts include key equivalence and union compatibility problems. Key equivalence problems result from two or more schemas identifying the same entity by means of semantically different keys. A union compatibility (schema isomorphism) problem between two entity types arises, when the domains of their attributes do not match. Special cases of this conflict include situations when a different number of attributes is used to describe entities that should be compatible, or when objects described by a set of attributes in one schema, are represented by a subset of those attributes in another. Rusinkiewicz and Czejdo (1987) introduce a generalized outer union operator is defined to deal with this problem.

Outside of these two classes are *abstraction level incompatibility* and *schematic discrepancy conflicts*. Abstraction level incompatibility refers to generalization and aggregation conflicts that make relating data in the two databases difficult, since it is not possible, in general, to map aggregated values onto the individual data by de-aggregating them.

Schematic discrepancy conflicts, addressed by Krishnamurthy et al. (1991), arise when data in one schema correspond to meta-data in another.

Recent work on heterogeneity reconciliation covers a wide spectrum of techniques. Uncertainty modeling and "instance level" conflict resolution have been addressed by Lim and Srivastava (1993, 1993b) and by DeMichiel (1989).

The use of semantic values and arbitrary conversion functions is proposed by Sciore, Siegel and Rosenthal (1992). Hammer and McLeod (1993) present a mechanism for resolving semantic heterogeneity in federations of autonomous databases that is based on meta-functions and local lexica utilizing a common knowledge representation. "Schema level" resolutions techniques address specific incongruities. Kim et al. (1992) use a classification of conflict resolution techniques to resolve many structural schema conflicts using a multidatabase language SQL/M. Similarly, a query language influenced by the Datalog paradigm is adopted by Krishnamurthy et al. (1991) to resolve schematic discrepancies. Common to most of these proposals is the acknowledged need for some form of meta-information, whose purpose is to describe *how* data integration is to be performed. The general term *mediators* has been used by Wiederhold (1991) to encompass the variety of tools for entity and object description that incorporate meta-information.

Our approach to the resolution of schema and data conflicts originates from the paradigm of the MSQL multidatabase language as defined by Litwin (1990), and has been influenced by the techniques described by Embley, Czejdo and Rusinkiewicz (1987), for an interactive query formulation in a multidatabase environment.

MSQL is an extension of SQL for a loosely coupled federations of database systems with relational interfaces. Multiple databases are visible to the MSQL user and attributes and tables from any of the databases may be referenced. The scope of a MSQL query is a collection of local relational schemas. A *multiquery* is a synthetic expression for a set of queries, one for each schema in the scope. Collective names, called *semantic variables*, can be used to refer to different identifiers in different schemas, thereby allowing the factorization of a single, abstract multiquery into a set of elementary queries. By considering the bindings between semantic variables and their corresponding real data items, the multiquery processor can map each semantic variable onto the local schemas, yielding a set of elementary, monodatabase queries. The results of these queries can be joined. *Interdatabase joins* represent a rudimentary form of data fusion, through which data retrieved from different sources can be explicitly combined.

While some features of MSQL make it potentially very expressive, the language is quite limited in its ability to deal with structural incompatibilities. Semantic variables represent a first step in the direction of structural abstraction, but their use is limited to the resolution of synonym conflicts on isomorphic schemas. Interdatabase joins are limited to domain-compatible attributes, to which standard relational operators can be applied without transformation of the operands. Furthermore, the language offers little help in case of data type discrepancies and union incompatibility. In general, the expressiveness of the language appears to be limited by the inability to provide the query processor with the meta-information needed to resolve schema conflicts.

The language described by Embley et al. (1987) adopts an approach to schema integration and query formulation with limited integration visibility. The notions of *connectors* among relations and of *extended abstract data types*, comprehensive representations of knowledge about data domains, are used to resolve a few, well-defined structural incompatibilities. Additional information is represented by the domain knowledge base, together

with some heuristics based on the interpretation of object identifiers, which help disambiguate implicit (i.e., undeclared and not explicitly described) relationships among given data items. The language features a generalized outer union operator to deal with union incompatibility conflicts. Connectors are used to resolve key equivalence conflicts and domain mismatch in joining attributes.

Current commercial database systems do not yet seem to fully address the problems of heterogeneous access. Oracle, Sybase and other products do allow connections to multiple database instances (e.g. by qualification of database objects in the query with instance names). However, to the best of our knowledge they do not attempt to perform query decomposition and conflict resolution.

3 SPECIFICATION OF MULTIDATABASE QUERIES

We propose to carry the MSQL paradigm further, by providing language features for the description, at the query specification level, of semantic relationships among data domains, along the lines described by Embley et al. (1987). Our approach, however, is not based on a domain-specific knowledge base, providing instead language support for the specification of additional data modeling concepts.

Semantic variables represent a first step towards explicitly representing the context surrounding the evaluation of a multiquery, by introducing new identifiers in the name space of database objects. The main limitation of this approach is that these identifiers are simply aliases that are only defined to the extent that they *refer* to existing database objects.

To improve upon this situation, we propose to introduce a new set of *virtual database objects*, as opposed to just *names*, giving a mechanism to relate them to existing local objects. In the current proposal, these objects are defined in particular as *attributes* of a new "global" entity –this can be thought of as a degenerate schema composed of only one entity– that can be referred to in multiqueries. They are typed to the same extent regular attributes are.

The main shift from the semantic variables viewpoint is that now global attributes exist *independently* of the local schemas. They are generally used to describe a particular database domain at a higher level of abstraction than the one of local objects.

The virtual objects paradigm is supported by a few features that are at the core of our multidatabase language, namely *global attributes*, *external function invocation*, *type coercion* and *implicit join completion*. Combined, these features should support both an abstraction mechanism, by allowing for the definition of higher-level entities, and a mapping mechanism, which describes operationally the semantic proximity among schema objects. Each of these features will be discussed in detail in this and in the next section.

Global attributes are used to represent collections of type-compatible expressions on local attributes. The expressions can include operators that are available at local database sites, either as built-in database functions (e.g. aggregation functions) or as user-defined applications with a well-defined interface to the database. We introduce a simple, two-level class hierarchy model, where the upper level includes the single global class in which all the higher-level attributes are defined. This class has one sub-class for each local schema, and each sub-class contains as attributes the complete set of attributes for the corresponding schema.

Mapping functions are used to define the transformation from local to global attributes, by mapping an expression that has local attributes as operands and uses the defined operators, onto a type-compatible global attribute. They also extend the definition of joins, both at the local and interdatabase level. *Generalized joins* are arbitrary predicates that may appear in the WHERE clause of a multi-query or an elementary query. Functions can be user- or DBA-defined and they can be implemented as external applications or scripts, as long as they comply with general rules for interfacing with the DBMS. Similar to connectors that may bridge the gaps across different schemas, functions offer a potential for expansion of the system to queries on external objects, that are beyond the expressiveness of relational algebra alone. Global attributes are typed, and their types are more general, according to a type lattice, than the type of each of the expressions on local attributes they represent*. *Type coercion* is necessary to preserve union compatibility across queries that span several local databases.

Implicit join completion, proposed by Litwin (1985), is a technique whereby some local joins can be left implicit in a query. Although it suffers from some known limitations, as mentioned in Section 3.5, it can still play a relevant role in our scenario, as the discussion in this paper will show.

3.1 A motivating example

The features identified in the previous section permit synthetic specification of multi-queries, in which global attributes can be used instead of, or in addition to, local attributes, generalized joins can be expressed, and some of the local joins can be omitted.

As an example, let us consider the following two relational database schemas, adapted from the presentation by Batini et al. (1986). Although both schemas represent library information about publications and their authors, they differ in number of relations, number of attributes, their domains, etc. Primary keys are underlined for each relation.

Example database schemas

Database for library *A*:
Relations BOOKS, TOPICS, UNIV describe books, keywords available for searching for the books, and the Universities whose libraries hold the books. The many-to-many relation between books and topics is modeled through the additional, linking relation BOOK_TOP. The many-to-many relation between books and universities is similarly modeled through the linking relation BOOK_UNIV.

BOOKS(b_code, title, author, bcost)
TOPICS(t_code, descr)
UNIV(name, state)
BOOK_TOP(b_code, top_code)
BOOK_UNIV(u_name, b_code)

*Basic types can be defined, for example, as INTEGER, REAL, {CHAR(n) for all n}, DATE, NUMBER, BOOLEAN and STRING=τ_{max} (the exact definition will be system-dependent). A simple type lattice is given by the relations INTEGER $\leq$ REAL, CHAR(i) $\leq$ CHAR(j) for $i \leq j$, REAL $\leq$ NUMBER and finally NUMBER $\leq$ STRING, DATE $\leq$ STRING, CHAR(n) $\leq$ STRING for all n.

Database for library B:
Similarly to the first database, here the many-to-many relation between PUBL (publications) and KEYW (available search keywords) is modeled through the linking relation P_K.

PUBL(code, title, author_fname, author_lname, cost)
KEYW(code, subj)
P_K(p_code, k_code)

Let us suppose that we would like to retrieve from both databases information about publications (books, journals, etc.) related to a particular topic, say 'xyz'. Assume that we want titles of the publications, their author(s), and their costs. Although structural and semantic discrepancies exist between the two schemas, they seem to be irrelevant to our query; intuitively we would like to write something like:

```
USE     A, B
SELECT _title_, _avail_from_, _author_, _cost_
WHERE match(_description_,'xyz');
```

This query specification is simple and natural: it indicates the scope of the search, specifies global class attributes to be retrieved and provides the retrieval condition. There are no references to local attributes, no specification of local joins, and the search condition is expressed using a high-level function. To evaluate this query, global class attributes must be mapped to their local schemas counterparts, separately for each local schema. The *match* predicate is defined as a virtual method at the global class level. Its implementation differs in general for each schema, to take care of the intricacies in local book subject identifier formats. Finally, the local joins need to be completed separately (and differently) for each local schema.

Obviously, to convert the above query into a set of standard (local) SQL queries, the query processor needs the meta-information about schema mappings and external functions. In the remainder of this section, we describe how this meta-information is supplied to the query processor, and give a relatively informal semantics of multiquery decomposition and execution. Missier (1993) gave a more formal definition of the language semantics, based on a variant of multi-relational algebra defined by Grant et al. (1991).

3.2 Global attributes and mappings

The mappings of local to global attributes involve local relations, built-in operators and user-defined functions to accommodate typical attribute-level conflicts. The implementation of user-defined functions is discussed in Section 4.

In the example, we must define the mapping from attributes in A.BOOKS and from attributes in B.PUBL onto _TITLE_, _AUTHOR_, _COST_. To describe these mappings and the semantics of multiqueries, we will use the following notation.

Let $\mathcal{S}$ be the set of all databases involved in the multiquery (the multiquery scope), $\mathcal{C}$ the set of global attributes, and for each $s \in \mathcal{S}$, let $\mathcal{A}_s$ be the set of all local attributes

for database s. Also, let $\mathcal{F}=\{f_i : \tau_1, \ldots, \tau_{n_i} \rightarrow \tau\}$ represent the collection of the mapping functions. For a given pair $s \in \mathcal{S}$, $c \in \mathcal{C}$ we define the *Mapping Descriptor*:

$$MD^s_c =< A, f, f' >,$$

where $A = \{A_1, \ldots, A_k\} \subseteq \mathcal{A}_s$, $f : type(A_1), \ldots, type(A_k) \rightarrow type(c) \in \mathcal{F}$ and $f' = f^{-1}$ if f is unary and invertible, undefined otherwise. Notice that MD is a partial function, i.e., it may be not defined for some c and s.

Examples of mapping functions include $concat(string, string) \rightarrow string$ and $Pounds_to_USD(real) \rightarrow real$ with its inverse $USD_to_Pounds(real) \rightarrow real$. The inverses of functions are useful in some simple cases when the rules for mapping resolution, described below, allow for "lexical simplification" of expressions.

Using this notation, the MDs for our example can be defined as follows (I indicates the identity function):

$$MD^a\text{_TITLE_} =< \{\text{BOOKS.TITLE}\}, I, I >$$

$$MD^b\text{_TITLE_} =< \{\text{PUBL.TITLE}\}, I, I >$$

$$MD^a\text{_AUTHOR_} =< \{\text{BOOKS.AUTHOR}\}, I, I >$$

$$MD^b\text{_AUTHOR_} =< \{\text{PUBL.AUTHOR_FNAME, PUBL.AUTHOR_LNAME}\}, concat >$$

$$MD^a\text{_COST_} =< \{\text{BOOKS.BCOST}\}, Pounds_to_USD, USD_to_Pounds >$$

$$MD^b\text{_COST_} =< \{\text{PUBL.COST }\}, Lit_to_USD, USD_to_Lit >$$

$$MD^a\text{_AVAIL_FROM_} =< \{\text{UNIV.UNAME}\}, I, I >$$

$$MD^a\text{_DESCRIPTION_} =< \{\text{TOPICS.DESC}\}, I, I >$$

$$MD^b\text{_DESCRIPTION_} =< \{\text{KEYW.SUBJ}\}, I, I >.$$

Figure 1 illustrates the relationship between local attributes from A and B and class $\mathcal{C}$. Solid lines delimits the class, while dotted lines indicate which local attributes are mapped into the global attributes.

3.3 Global Methods

In addition to the attribute mappings described above, for each database we must specify the meaning of "match" between an attribute and a constant string. This function is an example, however rudimentary, of a (virtual) method defined on the global class, for which each subclass has to provide a specialized version. These methods can take as arguments only attributes from the class to which they belong. This restriction allows a preliminary straightforward implementation of these methods simply as macros to be are expanded

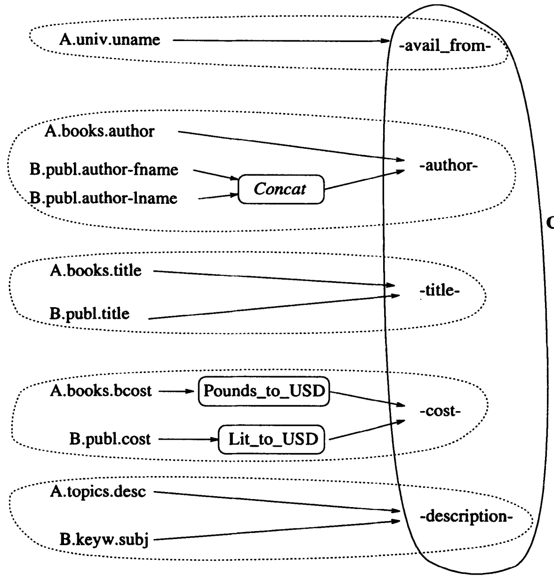

Figure 1 Local attributes, global attributes, and mappings

by the query interpreter. As such, static typing on the function arguments is currently not enforced. Assuming that the string comparison standards are different for each of the databases, the *match* predicate can be declared as follows:
For database A: "X LIKE Y" → match(X,Y);
For database B: "upper(X) = upper(y)" → match(X,Y);
(where "upper" is assumed to be a built-in string conversion function at B's site).

These functions are generally query-independent and extend the existing suites of built-in database functions.

3.4 Multiquery decomposition and interpretation

Our model for query decomposition borrows its general concepts from an earlier MSQL implementation by Suardi and Rusinkiewicz (1993), with a few important extensions.

In MSQL, a multidatabase query is decomposed into a set of elementary, monodatabase queries. Each query is submitted independently to the local DBMS systems. If needed, interdatabase joins are performed (at a central site) on the relations returned by the local queries. In first approximation, the decomposition process amounts to computing, for each Database Graph (DBG, see the next section for a definition) in the query scope, the largest connected subgraph that can be embedded into the DBG.

The main steps in our decomposition process are the following. First, references to global attributes and to class methods must be resolved. Second, *implicit join completion* is required to reconstruct the missing local joins. Finally, the various execution plans at the

local levels must take into account the invocation of user-defined functions, which appear at the system level as external applications. In this section we describe the resolution of global references.

The initial multiquery is replaced by a set of queries, one for each database in its scope (listed in the USE clause). In each query, the reference to a global method is substituted by its corresponding expression through a lexical macro expansion. If the inverse mapping function is defined, we can use it to simplify selection expressions of the form: $c < relop >< constant >$, where $MD^s_c =< A_1, f, f^{-1} >$, by replacing it with $A_1 < relop > f^{-1}(< constant >)$. For example, for database A the expression _COST_ > 100, where _COST_ = $Pounds_to_USD$(BOOKS.COST), would become simply BOOKS.COST > USD_to_Pounds(100), and the right hand side is now a constant expression.

We then use each new query independently to generate its "localized" version. For each database $s \in \mathcal{S}$ and each $c \in \mathcal{C}$ appearing in the SELECT or in the WHERE clauses of the global query, if $MD^s_c =< \{A_1, \ldots, A_k\}, f, f' >$ is defined, then expression $f(A_1, \ldots, A_k)$ is appended to the local SELECT list. If f is defined externally to the local DBMS, the actual SQL query that is sent to the database contains only the attributes $A_1, \ldots, A_k$ and the final local- and interdatabase- join computations are carried our by the global query processor. We discuss the implementation issues further in Section 4.

Since MDs are partial, for a given c and s, MD^s_c may be undefined. If c appears in the projection list of a global query, in a local query it is mapped into a dummy NULL_c attribute (of the same type as c's) to maintain union compatibility across elementary queries. If it appears in a selection condition (WHERE clause), then the local query is discarded as non-pertinent, as in Suardi and Rusinkiewicz (1993). If, in particular, the global WHERE clause involves an interdatabase join on such attribute, then the entire global multiquery is rejected.

At the end of this step we are left with a collection of queries, in the MSQL style except that some of the local joins may still be undefined. The result for our running example is shown below. Missier (1993) has a more complete discussion on type coercion through the application of library functions to single attributes.

```
USE A:
SELECT books.title, univ.name, books.author, Pounds_to_USD(books.bcost)
WHERE topics.descr LIKE 'xyz';

USE B:
SELECT publ.title, '', concat(publ.author_fname, publ.author_lname), Lit_to_USD(publ.cost)
WHERE upper(keyw.subj) = upper('xyz');
```

The specification of interdatabase joins is complicated by entity-domain conflicts. Generalized joins and externally defined predicates help overcome this problem. The strong typing of the whole system allows early detection of entity-domain incompatibilities.

In the next section we focus on the implicit join completion in elementary queries.

3.5 Implicit joins

In the multiquery of Section 3.1, no local joins appear. Since local joins correspond to particular access paths in the local schemas, and these paths generally differ from schema to schema, omitting them altogether makes the final query more concise and expressive. The cost of this convenience is the need for additional information about local schemas that must be provided to the query processor to perform automatic join completion.

Litwin (1985) applies the notion of *implicit joins*, previously used in the context of algorithms for the universal relation approach by Maier and Ullman (1983), to the multidatabase environment. The general idea is that if a collection of *connections* among attributes is defined a priori, then there exists an algorithm to determine, given a subset of those attributes, a set of "most reasonable" joins among them. This means that, given enough information about the schema (in Maier and Ullman (1983) functional dependencies are used), some of the joins can be left implicit in a relational query expression.

In the original formulation, the algorithm relies on the existence and automatic detection of *natural dependencies*. In reality, current database technology may allow additional schema information referential constraints to be stored in the data dictionary, but this practice is not even enforced. Therefore, we will conservatively consider only *user-supplied* dependencies, and introduce a declaration section in our language for the purpose (see Section 3.6).

The notion of automatic completion can be fruitfully applied in our context to achieve abstraction. Suppose that, for each single database, such a set of connections is given, for example as relationships between keys. Suppose further that for each database a completion algorithm can be run on a set of selected and projected attributes. Then the incomplete elementary queries yielded by our decomposition algorithm (Section 3.4) could be completed separately.

This approach addresses a problem of *schema isomorphism*: details of *how* relations in a local schema should be joined typically differ across the set of local schemas, resulting in different sets of joins for queries that are meant to retrieve *semantically related* data. By confining those details in a set of declared or deducible connections, we are able to separate them from the query, as proposed also by Miller et al. (1993). As a result, local queries expressed on non-isomorphic schemas can still be collected into one single multiquery. As should be clear, the mechanism of implicit joins can be fully exploited when coupled with the use of class attributes.

The main problem with this technique is that the set of (minimal) connections may not be unique. Intuitively, input information may not be sufficient to determine a unique *meaning* for each subset of the attributes and arbitrary selections. This problem has been addressed by Maier and Ullman (1983) by introducing the notion of maximal objects. The completion algorithm defined by Litwin (1985) takes the union of the resulting queries.

In our version we have chosen not to assume any predefined strategy for resolving ambiguities. Rather, if the query admits more than one completion, a request is issued to the user to remove the ambiguity. The chosen solution can then optionally be assumed as the default in subsequent queries, thereby reflecting the actual user's view in a particular context.

The input to the completion algorithm is composed of two parts. The first part is query-independent and expresses user-supplied or derived dependencies in the form of a *database graph* (DBG). In a DBG, nodes represent database relations. An edge $\{<$

$R.A_i, op_i, S.B_i >\}$ for $1 \leq i \leq n$ represents the predicate $\bigwedge_{i=1}^{n}(R.A_i < op_i > S.B_i)$, while an edge of the form $r(R.A_1, \ldots, R.A_k, S.B_1, \ldots, S.B_h)$, where r is an external $k+h$-ary predicate, represents a generalized join between relations R and S. The second part of the input, the *query graph* (QG), represents a specific query and is constructed *by the query processor*. The nodes of the Query Graph correspond to relations referenced by the query and the edges represent join conditions of the query.

For our simple example databases, the DBGs can be derived directly from the schema referential constraints. They are defined by listing the graph edges corresponding to the join conditions:

For database A:

BOOKS.B_CODE = BOOK_TOP.B_CODE;
BOOK_TOP.B_CODE = TOPICS.T_CODE;
BOOKS.B_CODE = BOOK_UNIV.B_CODE;
BOOK_UNIV.B_CODE = UNIV.NAME;

For database B:

PUBL.CODE = P_K.P_CODE;
P_K.K_CODE = KEYW.CODE;

The two Database Graphs are depicted in Figure 2.

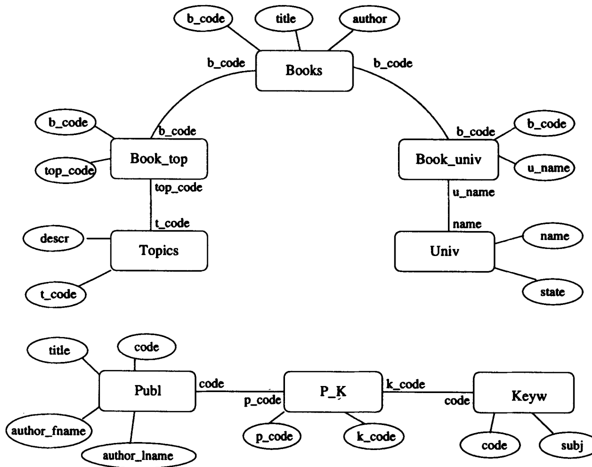

Figure 2 Database graphs for the library databases

Using the completion algorithm, we obtain the following queries that can be submitted to the local systems:

```
Database A:
SELECT books.title, univ.name, books.author, Pounds_to_USD(books.bcost)
FROM books, topics, book_top WHERE topics.descr LIKE 'xyz'
AND books.b_code = book_top.b_code

Database B:
SELECT publ.title, '', concat(publ.author_fname, publ.author_lname), Lit_to_USD(publ.cost)
FROM publ, p_k, keyw WHERE upper(keyw.subj) = upper('xyz')
AND publ.code = p_k.p_code
AND keyw.code = p_k.k_code
```

3.6 Declarations in a multidatabase language

As the example in section 3.1 shows, a high-level multiquery is defined with respect to a *context*, defined by the domain-specific information needed to perform the mappings. This includes the global class definition with its attributes and virtual methods, the specialized method implementation, the external function declarations and the schema Database Graph for implicit joins completion. Changing the context alters the meaning of the query. We propose to introduce declaration sections as wrappers for all context information. Contexts can be referred to and scope rules for contexts are defined to associate each query with its appropriate context†.

3.7 Example of Declaration Section

Declarations can be compared to generalized views that reflect *one* of the possible abstractions of the underlying data. Declarations resemble the federated schemas of Sheth and Larson (1990) in that they are superimposed on the local schemas, but they can be replaced, totally or partially, by the user, depending on the needs of each specific query.

Here we present a sample declaration section that contains all the elements necessary for the query processor to analyze and execute our example multiquery. The syntax is the one used in our prototype implementation, and should not represent an issue at this point.

```
class library
              title_: string,
              avail_from_: string,
              author_: string,
              cost_: real,
              description_: string
end;

maps
              concat(string, string): string for B;
```

†In the final section of this paper we briefly discuss the implications of defining different contexts that can be applied to the same query.

```
                    Pounds_to_USD(real): real for A;
                    Lit_to_USD(real): real for B;
end;

dbstruct for A
    dep
                    books.b_code = book_top.b_code;
                    book_top.top_code = topics.t_code;
                    books.b_code = book_univ.b_code;
                    book_univ.u_name = univ.name;
    members
                    books.title: title_;
                    books.author: author_;
                    Pounds_to_USD(books.bcost): cost_;
                    univ.name: avail_from_;
                    topics.descr: description_;

end;

dbstruct for B
    dep
                    publ.code = p_k.p_code;
                    p_k.k_code = keyw.code;
    members
                    publ.title: title_;
                    concat(publ.author_fname, publ.author_lname): author_;
                    Lit_to_USD(publ.cost): cost_;
                    keyw.subj: description_;

end;
```

There three main sections in the fragment above: **class**, **maps** and a sequence of **dbstruct** sections, one for each schema in the scope of a potential query. The **class** section declares the global typed attributes. The **maps** section declares the user-defined functions used in the local to global mappings. In general, user-defined predicates used in the generalized joins are also declared here. Each function may be qualified with a "for <schema>" clause, meaning that a corresponding implementation must be available when that schema is accessed. An unqualified function is available everywhere (possibly with different implementations at different sites).

The **dbstruct** section consists of two parts: a **dep** subsection[‡], where the DBG for the schema is declared, and a **members** section, that lists the correspondences of local to global attributes. The functions declared in the **maps** section can be referenced here.

These declarations syntactically precede a query and define its context. By changing the declarations, the same query may assume a different semantics. In particular, in the sections above exactly one DBG is defined for each schema. Changing the DBG may alter the way implicit joins are resolved.

[‡]This keyword is derived by the MSQL idea of "natural dependencies".

3.8 Usage

A hybrid architecture emerges from this approach, in which static schema integration is mixed with a "pure" multidatabase manipulation language. Typically, a MDBA would be in charge of creating abstractions and views and of declaring and defining libraries of external functions available at some local sites.

Users may define a context by choosing one from a library of available declarations (using #include style directives). Following this basic choice, some sections can then be overridden, excluded or *specialized*[§], in much the same way as one would with a class library in an object-oriented model.

It should be noticed that schema integration in this architecture is neither completely hidden nor entirely visible (see Heiler et al. (1991) for a definition of integration visibility). It is in part performed *a priori*, but there is no real global schema, and users can choose to access the system at different levels. For instance, the declarations introduced above also support the following query:

```
USE    A, B
SELECT books.bcost, publ.cost
FROM   books, publ
WHERE  books.title = publ.title;
```

In this query, global attributes are bypassed altogether and the more traditional SQL format (with the FROM clause) is used, under the assumption that the attributes to be joined are compatible with respect to the standard equijoin operation.

4 QUERY PROCESSOR IMPLEMENTATION

A prototype implementation of the proposed language was carried out in the execution environment developed at the University of Houston, called Narada, described and implemented by Halabi, Ansari and others (1992). Narada facilitates the specification and execution of multi-system applications in a distributed and heterogeneous computing environment. It includes a communication layer for inter-task communication, a number of local "agents" that act as proxy users and can execute tasks at the individual connected sites, and a scripting Distributed Operation language, DOL, for the specification of tasks, their coordination requirements and inter-task data flow. DOL is the target language for the prototype MSQL compiler developed at University of Houston. The basic multiquery execution strategy for MSQL is described by Suardi and Rusinkiewicz (1993) and is a basis for our implementation.

In this section we present a query execution schema that includes the invocation of external processes for generalized joins and describe the algorithm for join completion.

[§] Currently, our grammar lacks the capability of expressing overriding declarations.

4.1 Multiquery evaluation

Two basic types of external references are allowed in a multiquery: *filters* that appear in joins of the form $f(R.A_{i_1}, \ldots, R.A_{i_n}) < op > g(S.B_{i_1}, \ldots, S.B_{i_m})$, and *predicates* that appear in selections of the form "... WHERE ... $p(R.A_{i_1}, \ldots, R.A_{i_n})$". Both filters and predicates can be evaluated in a tuple-at-a-time fashion on an input relation. Filters transform relations into new relations, preserving their cardinality but altering their structure, whereas predicates return a subset of the input relation (the tuples that satisfy the predicate), unaltered. Awk scripts and "grep-like" commands provide a good intuition for shell level filters and predicates, respectively.

In the following, we assume for the purpose of illustration that filters and functions can be invoked by a system shell, and that relations obtained as a result of queries are available as text files, one line for each tuple¶. We indicate the conversion to and from the textual representation by the functions *db_to_file*(*SQL_query*, *file*) and *file_to_db*(*file*, *DB_table*). The evaluation and effects of filters and predicates is shown in Figure 3. A filter f(A,B) applied to table R(ABC) requires first the selection of columns A,B from table R into a file. The filtered output is placed in the resulting relation S into column S.F, along with the corresponding values of C. Notice that we cannot assume column C is carried through f. Similarly, a predicate p(A,B) applied to relation R(ABC) requires exporting R.A, R.B columns into the file used to evaluate the predicate. The rows corresponding to the qualifying tuples are then inserted in the resulting table S.

Consider the evaluation of predicate $p(S)$, where $S = \pi_{A_1,\ldots,A_k}(R(A_1, \ldots, A_m))$. Since in general S does not include the primary key for R, after applying p the qualifying tuples must be joined with the original relation R:

$$R' = (p(S) \bowtie_{A_1,\ldots,A_k} R).$$

Operationally, this expression translates into the following sequence of elementary tasks: apply *db_to_file*(*SQL_query*, *f_in*), where *SQL_query* retrieves S; apply p to *f_in* resulting in *f_out*; invoke *file_to_db*(*f_out*, *TMP*); join R with temporary relation TMP. In a DOL implementation, the first task would issue the SQL query and pass the resulting relation to the second task, which would call p as a shell script and pass its results to the third task for the final join.

The generalization to the case of multiple tables involved in p is straightforward. In this case, however, there is more room for optimization of the resulting task sequence.

The main problem with filter application is that, unlike for predicates, we cannot join the temporary table resulting from the evaluation of a filter with the original table on existing attributes, since the original attributes are mapped into new ones. This problem can be obviated by introducing auxiliary keys that are guaranteed to survive intact the filtering process. Pseudo-columns, normally available in commercial SQL implementations, can be used for the purpose. If we assume that the filtering process does not alter the row sequence, row numbers can be used as auxiliary keys. Suppose filter $f(S)$, where

¶This simplifying assumption is by no means essential, as more efficient ways to pass data across functions can be devised.

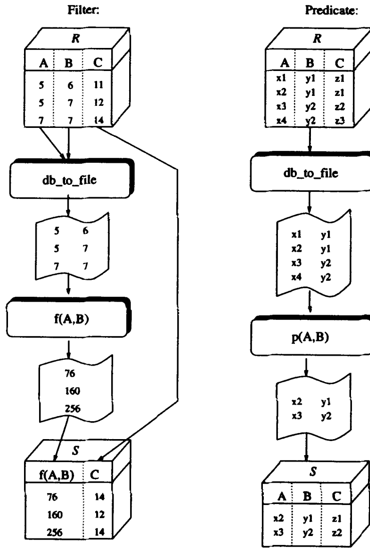

Figure 3 Effect of filter and predicate evaluation

$S = \pi_{A_1,\ldots,A_k}(R(A_1,\ldots,A_m))$, is to be evaluated. If we assume that pseudo-column RN is available for each relation, the resulting relation can be computed simply as:

$$R' = (f(S) \bowtie_{RN} R).$$

Like predicates, filter computations can also be implemented in DOL scripts.

4.2 Implementation of Join Completion

The completion algorithm we have implemented uses the database graph (DBG) and the query graph (QG), defined in section 3.5. The DBG is built from user declarations and does not change for queries that refer to the same declarations. The QG is superimposed on the DBG, by marking nodes and edges and adding new edges when necessary, as follows:

- for each attribute in the projection or in a selection, the node corresponding to the relation to which that attribute belongs is marked;
- for each (generalized) join, the edge representing that join in the DBG is marked, if it exists. Otherwise, a new edge labeled with the attributes in the join is created to

connect the relation nodes[||]. For a join of the form $R.A < op > S.B$, there will be an edge between nodes R and S, labeled with A, op and B. Notice however, that a collection of edges between the same two relations, $\{R.A_i < op_i > S.B_i\}$, is represented by a *single* edge with label $\{< R.A_i, op_i, S.B_i >\}$. Such edge could typically represent a multiple join involving partial keys in two relations. This representation guarantees that at most one edge resulting from explicit joins exists between any two nodes. The same kind of labeling also accommodates relations of the form $r(R.A, S.B)$.

The marked nodes and edges define a component in the DBG and the goal of completion is to make this component connected, by adding non-marked edges from the embedding DBG. The set of minimal-depth trees connecting a given subgraph component yields a minimal number of additional necessary edges. The algorithm computes the set of spanning trees for the whole graph, using a binary representation similar to the one shown by Litwin (1985). Since all the spanning trees have to be explicitly represented in this step, it is essential that their representation be compact. The trees are then pruned to dispose of non-marked nodes. Minimal trees are then selected from the surviving set.

Each edge in the resulting tree now represents a join. The conjunction of the set of joins so generated is the required completion for the query. Notice that adopting a single-edge representation for multiple joins between the same two relations ensures that standard algorithms for spanning trees can be used.

Multiple completions resulting from multiple minimal spanning trees correspond to different query *interpretations*. Litwin proposes to resolve this ambiguity by considering the *union* of the resulting queries. While this approach can be reasonable in some instances, we feel that its correctness in the general case is questionable. A more conservative approach is, of course, to let the user decide on a case by case basis, with the option of using the decision, as the default choice for future reference. A slight refinement of this strategy is to present the user with a choice among only those edges which, when included in the query, would minimize the set of remaining minimal trees. Choosing among the edges which are not marked and are *not* shared by different trees, seems to be a good heuristics[**].

5 CONCLUSIONS

The work described in this paper can be viewed both as a proposal for enhancements to a well-known MSQL approach to multidatabase manipulation, and as an indication of the limits imposed by making only structural external knowledge (meta-data) available to the query processor.

In this final section we analyze our approach with respect to the types of conflicts that can or cannot be resolved by the proposed mechanisms, and point out some directions for further work.

[||] Note that, in this case, generalized joins are limited to binary joins. The generalization to arbitrary n-ary joins requires adapting the completion algorithm to work on hypergraphs.

[**] The tree representation may have to be translated back into a human-readable representation of the corresponding query by a front-end module before being presented to the user.

5.1 Conflict resolution

A *synonym* conflict is resolved by introducing a class attribute c as the common name for two synonyms R.A, S.B. The mappings R.A $\rightarrow c$ and S.B $\rightarrow c$ are, in the simplest case, identities. *Homonyms* can be avoided by simply using, as in MSQL, full object qualification (< *database* > . < *table* > . < *attribute* >). In case of *data type* conflicts, a new global attribute of a more general type (the join of the two types in the lattice) can be used. In this scheme, type casting works only from lower to upper types. Notice however, that specific type conversion functions, either built-in or externally defined, can be explicitly invoked, for instance to convert a date from a string representation *back* into a different machine-dependent data type. The dynamic columns proposed for MSQL resolve *data scaling* problems. Expressions including external functions generalize dynamic columns. To account for *data precision* conflicts, Sheth and Kashyap (1992) propose to use an external function to define a many-to-one mapping from the more precise to the coarser attribute, as in the case of a conversion table from numerical marks to letter grades.

Key equivalence conflicts are alleviated by a combination of class attributes and functions and implicit joins completion. Even when no common key can be found, it may be possible to define a single multiquery where the local joins in the elementary queries are implicit, and then let the system do the navigation on each database graph separately. Likewise, semantically similar selections may differ for the local attributes, as in Example 3.1 for the book topics. In this case, class methods can be used to hide the differences. Class attributes together with type casting could be used to preserve *union compatibility* and to deal with *missing attribute* conflicts.

Implicit joins completion should obviate the problems caused by *non-isomorphic local schemas*. External functions account for differences in the number, and possibly types, of attributes.

Some conflicts have not yet been addressed in this work. These include *schematic discrepancies*, *missing data items*, *attribute integrity constraints* and *default value conflicts*. Krishnamurthy et al. (1991) propose to resolve schematic discrepancies by introducing higher-order variables whose domains are extended to meta-data. Missing data items and default values require specific domain-knowledge, which could find a place in an appropriate declaration section. Attribute integrity constraints are also context-dependent. In general, our proposal does not address any conflicts that require instance level solutions.

5.2 Further Work

The concepts presented in this paper have been formalized to some extent by describing an extension to relational algebra which encompasses external functions and classes. Details of this formalization, as well as of directions for further work summarized here, can be found in the work by Missier (1993).

The multidatabase language approach presented here can be expanded in several directions. The implementation of mechanisms for the invocation of external programs should be made more general. External applications are presently very rigid in their structure, due to the assumptions we made to ensure they can be interfaced with the query processor without using SQL as an embedded language, and having to write *ad hoc* database access interfaces. A whole collection of class declaration sections should be available. To establish a context, user should be allowed to simply refer to a section, or to *define* a whole new sec-

tion, or to *modify* an existing section. In general, a hierarchy could be imposed on classes, in two distinct ways: across a set of declarations (hierarchy of declaration sections), and within a single declaration (multi-level class structure for a single context). The attribute name resolution algorithm needs to be adapted to accommodate the second possibility, since now *repeated* substitutions of class attributes would be allowed.

Finally, further study is needed to overcome the limitations of the join completion approach. The idea of using *weighted* database graphs to represent extra information about the *meaning* of a join is worth investigating. The implications of extending the algorithm to allow for implicit *interdatabase* joins also need, in our opinion, to be better understood.

ACKNOWLEDGMENTS

The authors would like to thank W. Litwin and W. Jin for their valuable comments.

REFERENCES

Ahmed, R., De Smedt, P., Du, W., Kent, W. et al. (1991) The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24(12):19–27.

Batini, C., Lenzerini, M., and Navathe, S.B. (1986) A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4).

DeMichiel, L. (1989). Performing operations over mismatched domains. In *Proceedings of the Fifth IEEE International Conference on Data Engineering*, pages 36–45. IEEE Computer Society, IEEE Computer Society Press.

Embley, D., Czejdo, B., and Rusinkiewicz, M. (1987) An approach to schema integration and query formulation in federated database systems. In *Proceedings of the Third International Conference on Data Engineering.*

Grant, J., Litwin, W., Roussopoulos, N., and Sellis, T. (1991) An algebra and calculus for relational multidatabase systems. *IEEE-IMS 1991, Proceedings of the First International Workshop on Interoperability in Multidatabase Systems.*

Lim, E.-P., and Srivastava, J. (1993) Attribute value conflict in database integration: An evidential reasoning approach. Technical Report TR 93-14, University of Minnesota, Dept. of Computer Science.

Lim, E.-P., Srivastava, J., et al. (1993) Entity identification in database integration. In *Proceedings of the Ninth IEEE International Conference on Data Engineering*, pages 294–301, 10662 Los Vaqueros Circle, POB 3014, Los Alamitos, CA 90720-1264, April 1993. IEEE Computer Society, Austrian Computer Society, IEEE Computer Society Press.

Halabi, Y., Ansari, M., Batra, R., Jin, W., Karabatis, G., Krychniak, P., Rusinkiewicz, M., and Suardi, L. (1992) Narada: An Environment for Specification and Execution of Multi-System Applications. In *Proceedings of the Second International Conference on Systems Integration.*

Heimbigner, D., and McLeod, D. (1985) A federated architecture for information management. *Proceedings of the IEEE International Conference on Data Engineering*, 3(3).

Hammer, J., and McLeod, D. (1993) An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems. *International Journal of Intelligent and Cooperative Information Systems*, 2(1):51–83.

Heiler, S., Siegel, M., and Zdonik, S. (1991) Heterogeneous information systems: Understanding integration. In M. Rusinkiewicz Y. Kambayashi and A. Sheth, editors, *IMS '91*, pages 14–21. IEEE Computer Society, IEEE Computer Society Press.

Kim, W., Choi, I., Gala, S., and Scheevel, M. (1992) On resolving schematic heterogeneity in multidatabase. Technical report, UniSQL, Inc., Austin, TX.

Krishnamurthy, R., Litwin, W., and Kent, W. (1991) Language features for interoperability of databases with schematic discrepancies. In J. Clifford and R. King, editors, *Proceedings of ACM SIGMOD Conference*, pages 40–49. ACM.

Kim, W., and Seo, J. (1991) Classifying schematic and data heterogeneity in multidatabase systems. *IEEE Computer*, 24(12):12–18.

Litwin, W. et al. (1990) MSQL: A Multidatabase Language. *Information Sciences*, 49(1-3):59–101.

Litwin, W., and Abdellatif, A. (1986) Multidatabase interoperability. *Computer*, 19(12).

Litwin, W. (1985) Implicit joins in the multidatabase system MRDSM. In *Procs. IEEE-COMPSAC*, Chicago.

Maier, D., and Ullman, J. (1983) Maximal objects and the semantics of universal relation databases. *ACM Transactions on Database Systems*, 8(1):1–14.

Miller, R.j., Ioannidis, Y.E., and Ramakrishnan, R. (1993) The use of information capacity in schema integration and translation. In D. Bell R. Agrawal, S. Baker, editor, *VLDB*, pages 120–133. Morgan Kaufmann Publishers.

Missier, P. (1993) Extending a multidatabase language to resolve schema and data conflicts. Master's thesis, Department of Computer Science, University of Houston.

Missier, P. (1993) Extensions of MSQL: notes on an implementation. Internal Report, University of Houston, Dept. of Computer Science.

Missier, P., Rusinkiewicz, M., and Silberschatz, A. (1995) Providing multidatabase access – an association approach. In *Proceedings of the Sixth Workshop on Database Interoperability*, Hong Kong.

Rusinkiewicz, M., and Czejdo, B. (1987) An approach to query processing in federated database systems. In *Proceedings of the Twentieth Hawaii International Conference on System Sciences.*

Sciore, E., Siegel, M., and Rosenthal, A. (1992) Using semantic values to facilitate interoperability among heterogeneous information systems. *Transactions on Database Systems*, (12).

Sheth, A., and Kashyap, V. (1992) So far (schematically) yet so near (semantically). In *IFIP TC2/WG2.6 Conference on Semantics of Interoperable Database Systems.* Elsevier Scientific Publisher B.V.

Sheth, A., and Larson, J. (1990) Federated Databases: Architectures and Integration. *ACM Computer Surveys.*

Suardi, L., Rusinkiewicz, M., and Litwin, W. (1993) Execution of extended multidatabase SQL. In *Proceedings of the 9th IEEE International Conference on Data Engineering.*

Wiederhold, G. (1991) Mediators in the architecture of future information systems. *Computer*, 25(3):38–49.

Questions & answers

Question [Arnon Rosenthal]:

This language assumes that your system is loosely coupled. But customers building large systems don't like to have to use a different language for loosely coupled systems than for more tightly coupled ones. Couldn't you make it more SQL-like so that programmers could just use this?

Answer [Paulo Missier]:

This is a strict SQL extension, a superset of SQL. In particular, what is generated from an MSQL query, and sent to the SQL server, is just SQL.

Question [Arnon Rosenthal]:

Would it make sense to have the notion of equivalent attributes even within a single database?

Answer [Paulo Missier]:

Yes, this could be used within a single database.

Question []:

Are declarations established before every query? Or over a given session?

Answer [Paulo Missier]:

My personal idea is that it you should declare over a whole session. A single query is not enough. A transaction unit would be slightly better, but a whole application usually makes the most sense. It really depends on the individual application, however.

Question[]:

For functions such as concatenation, how are they defined? By a program, or...

Answer [Paolo Missier]:

Think of them as arbitrary functions that are available locally and can be exported. You can view them as local resources.

Question []:

Specified by programs?

Answer [Paolo Missier]:

Yes.

Question [Ling Liu]:

How do you manage a mapping involving, say, hundreds of databases, when you have no knowledge base, and no authority?

Answer [Paolo Missier]:

Well, even though there are many, you only need do them one at a time. Furthermore, you are not limited to using their vocabulary, you can define your own, or you can tweak existing declarations. Also, you can build on top of existing declarations, and hence generalize from what is already there.

Question [Ling Liu]:

What schema does the user need to use to compose a query, the top level or the lower ones?

Answer [Paolo Missier]:
Any level can be used. All can be accessed and referred to at query time.

Question [Amit Sheth]:
If the user looks only at the highest level of declarations in order to make a query, that is equivalent to schema integration, right?

Answer [Paolo Missier]:
Yes, that is true. However, there is no constraint forcing the use of only the highest level. The user can access the local declarations as well. Plus, here there is no static processor doing the mapping, so this does not correspond to classical schema integration.

Comment [Amit Sheth]:
Yes, but with static mapping you do it once and use it again and again. Here, you must do a mapping every time you start a new session or query.

Response [Paolo Missier]:
Yes, that is correct.

Question [Arnie Rosenthal]:
Optimal cost function for query completion is disasterous. The only optimal cost function is to match what is in the user's mind. Would anyone want to rely on join completion? Is it safe?

Answer [Paolo Missier]:
Join completion is important to this technology. In order to make it safe, you must go back to the user to aid in disambiguation.

Comment []:
Approaches based on join completion and other query disambiguation don't work in general. You must have user interaction based on the schema integration phase, not queries.

Response [M. Rusinkiewicz]:
The process of providing declarations can be viewed as schema integration, if you like. This integrated schema is interpreted, though, so it can be changed for the next query, if the user wants. Now, join completion, if the completion of a join is unique, is fine. If it is not unique, nothing here prevents you from declaring a join! Just like in object-oriented approaches...

Comment []:
Right. That is what you have to do. You cannot rely on automatic join completion.

Comment [Michael Siegel]:
This discussion is ten years old!!! If you can't figure it out, you ask the user!

Comment [Everyone]:
Right! Let's move on!