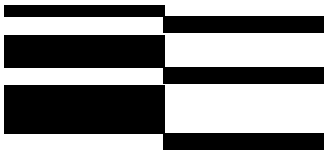
1



Multidatabase Languages

Paolo Missier[1]
Marek Rusinkiewicz[2]
W. Jin[3]

Introduction

Database systems based on SQL are well suited for homogeneous databases – either centralized or distributed. Most traditional database architectures, however, seem inadequate to handle different types of heterogeneity. Interoperability at the system level can be achieved to some degree by interposing an additional interface layer between a database system and the application, as in the ODBC solution [Mic94] and, more recently, in the analogous, Java-based JDBC proposal [HC96]. Other vendor-specific solutions provide network and protocol transparency by standardizing their SQL interface.

The problem of *data*, or *semantic*, *heterogeneity*, however, still remains. Different systems that own different pieces of data may come into conflict when they need to agree, at least in part, on the *meaning* of each other's data. This situation is common in loosely coupled database federations, where private data from a common domain of discourse is shared, and yet each local system insists on maintaining its ownership, autonomy and local views over its own portion.

In this chapter, we focus on the multidatabase access and manipulation language aspect of semantic interoperability. The proposed solutions we present are interesting in their common attempt to provide an expressive access language that can be used to describe and resolve semantic heterogeneity

conflicts. Their common approach is to extend standard access languages, such as SQL, with features to describe some of the semantics of the data being accessed (meta-data) – the design of SQL3, still an on-going process [Mel96], does not seem to address semantic conflict resolution.

One of the first attempts in this direction is described in [L+90]. MSQL introduced *multiple identifiers* and *semantic variables* to facilitate writing multidatabase queries. However, its ability to resolve semantic heterogeneity conflicts is limited. MSQL was later extended into MSQL+, with features to define "multidatabase objects" and their mapping to the local database objects [MR95]. Both MSQL and MSQL+ are discussed in detail in Section 1.0.7.

The area of multidatabase access with conflict resolution has been receiving increasing attention, and many more projects are under way than we can cover in this chapter. Among them are the InfoSleuth project at MCC, InfoHarness/InfoQuilt [SSKT95a, SSKT95b, SSKS95, KSS95], TSIMMIS [PGMU96], and Garlic [Car94]. Most of these systems use extensions of SQL as their information access language, in an attempt to preserve the declarative nature of the query language.

The rest of this chapter is organized as follows. In Section 1 we present a classification of semantic heterogeneity and proposed techniques for con-

flict resolution. Section 1.0.4 covers some history and basic design issues for multidatabase access languages. In Sections 1.0.7 we discuss MSQL and its extensions. We conclude the chapter in Section 1.0.11, with an introduction to updates in multidatabase systems, and in MSQL+ in particular.

Classification of Semantic Heterogeneity

Heterogeneity in multidatabase systems refers either to *system* differences, e.g. different DBMS, operating system architectures or networking protocols, or *semantic* differences, that refer to the different way similar real world entities are modeled. In this section, we concentrate exclusively on the latter. We offer a definition and classification of semantic heterogeneity and give an overview of heterogeneity resolution methodologies described in the literature.

1.0.1 Semantic Heterogeneity

Semantics can be broadly defined as "the scientific study of the relations between signs and symbols and what they denote or mean" [Woo85]. *Semantic heterogeneity*, in particular, refers to differences in the meaning and use of data that make it difficult to identify the various relationships that exist between similar or related objects in different components [HM93].

1.0.1) to schemas. Tables `SIGMOD.Conf` and `.AI_IA.Conferences` are intended to model similar information in different ways. Conflicts include (`conf_ref` vs. `name`) and type definitions (`integer` vs. `string(3)`), as well as modeling different fragments of the real world entities (the conference venue is not modeled at all in `AI_IA`).

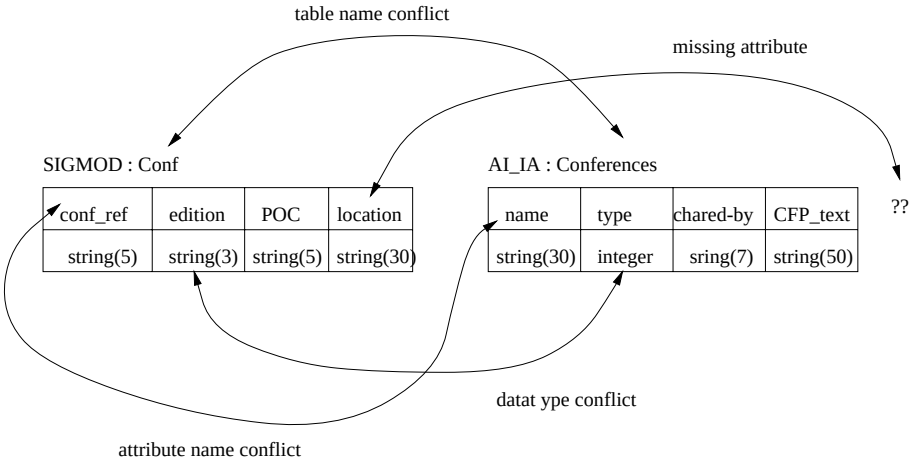


FIGURE 1.1

Semantic Heterogeneity Example

Hammer and McLeod [HM93] introduce the following levels of abstraction to define the spectrum of possible semantic heterogeneity:

1. **Meta data language**: Each local DBMS may use different data models and, consequently, different Data Definition sublanguages. Two such choices may be describing an ER model using the SQL DDL, as opposed to adopting a functional DB Language to describe a functional model.

2. **Ontology/Terminology**: Discrepancies occur at the ontological level.

3. **Meta-data specification**: Conflicts exist in the conceptual schemas.

4. **Object Comparability**: Equivalent/related objects cannot be easily identified.

5. **Low level data format**: Mismatching data types at the attribute level in corresponding database objects.

6. **Tools**: Each site may use a different DBMS, offering different system features.

The problem of semantic heterogeneity can thus be described as one of integrating "structurally dissimilar, but semantically equivalent" objects and of determining semantic equivalence of heterogeneous schemas. Although no widely accepted, fully automated techniques exist to accomplish this, we believe that database languages can and should be provided with the additional

expressivity required to assist in the systematic analysis and resolution of semantic heterogeneity.

1.0.2 Summary of Semantic Heterogeneity Conflicts

A classification of the most common heterogeneity conflicts is a good starting point for understanding the expressive power required in a federation-wide access language.

In [SK92][4], the authors describe a measure of distance between entities in different schemas, called *semantic proximity*, and apply it to the analysis of the relationship between semantic and structural heterogeneity. In this section, we present an overview of the kinds of structural conflicts identified in [SK92].

These conflicts can be broadly divided into the two classes of **Domain definition** and **Entity definition** incompatibility. Domain definition conflicts include naming (synonyms and homonyms), data type, data scaling, data precision, default value, and attribute integrity constraint problems. Entity definition conflicts include key equivalence, union compatibility, schema isomorphism, and missing data item problems. Outside of these two classes are abstraction level incompatibility and schematic discrepancy conflicts.

In the following, we briefly describe some of these conflicts. **Synonyms** occur when a set of semantically equivalent objects carry different names.

[4] A different classification is offered in [KS91].

Conversely, **homonyms** are semantically unrelated objects which incidentally carry the same name. **Data type** conflicts occur when equivalent objects have different types. Using different units of measure to describe similar objects, as when prices are expressed in different currencies, results in a **data scaling** conflict. Similarly, **data precision** conflicts refer to the use of different granularity for equivalent entities in different schemas.

Key equivalence problems result from two or more relations modeling the same entity by means of semantically different keys. Since a common key is not available, this conflict makes it difficult to retrieve data from the different entities using a single (multiple) query. Reconciliation, when possible, may be achieved through some forms of structural abstraction.

A **union compatibility** problem between two relations is generated when the number or the domains of their attributes do not match, or alternatively, when a one-to-one mapping among the respective sets of attributes does not exist. In [RC87], a generalized outer union operator is defined to deal with this problem.

Schema isomorphism conflicts refer to the different *number* of attributes used to describe entities which are schematically similar. A typical example is represented by a NAME attribute in one entity, which is split into LASTNAME and FIRSTNAME in the other.

Missing data item conflicts arise when objects described by a set of

attributes in one schema are represented by just a subset of those attributes in another. However, sometimes the values of missing attributes can be *deduced* through an inference mechanism or assumed as a default. For instance, the value of a TYPE attribute for table GRAD-STUDENT can be assumed to be "Graduate" and thus matched with an explicit corresponding attribute for a STUDENT table.[5]

Abstraction level incompatibility refers to *generalization* and *aggregation* conflicts. As an example of generalization, consider the entity PUBLICATION which can be represented in two databases by the table PUBL(PUBL#, AUTHOR, TITLE, ...) and the two tables BOOK(ISBN, AUTHOR, TITLE, ...) and JOURNAL(ISSN, AUTHOR, TITLE, ...), respectively. The first schema defines the same entity at a more general level of abstraction.

As an instance of aggregation conflict, consider the case of a database listing summary characteristics of a collection of books, vs. one of the two publication databases mentioned above. Data in the two databases cannot be easily related since, from the first, it is not possible to map values onto the second by de-aggregating the data.

Schematic discrepancy conflicts arise when data in one schema corresponds to meta-data in another. Resolution of schematic discrepancies generally requires a language, such as the one described in [KLK91], that allows

[5] This example is taken from [SK92].

references to data and meta-data to be mixed in one specification.

1.0.3 Semantic Heterogeneity Resolution Methodologies

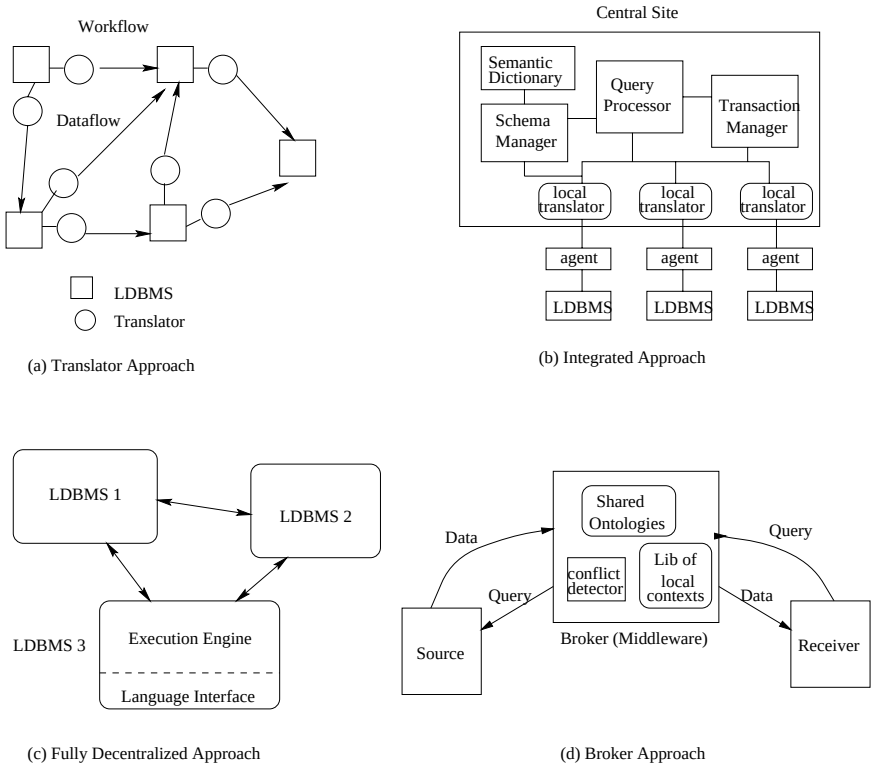


FIGURE 1.2

Semantic Heterogeneity Resolution Methodologies

Recent work in heterogeneity reconciliation covers a wide spectrum of techniques. Uncertainty modeling and "instance level" conflict resolution have

been addressed [EPS93a, EPS+93b, DeM89]. The use of semantic values and arbitrary conversion functions is proposed in [SSR92]. "Schema level" resolutions techniques address specific incongruities. For instance, a query language influenced by the Datalog paradigm is adopted in [KLK91] to resolve schematic discrepancy conflicts.

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 [Wie92] to encompass the wide variety of tools used for entity and object description that incorporate forms of meta-information.

In this section, we consider four broad approaches to the resolution of semantic heterogeneity in multidatabase systems: translation, integrated, fully decentralized, and broker-based.

Translation Approach. This approach is commonly used in a workflow scenario, where the flow of data among tasks involves access to several LDBs (see Figure 1.2-(a)), and the datapaths are known in advance. In addition, the LDBs schemas should be stable and well-known; these requirements apply well to corporate environments where some data-intensive business processes are being automated using workflow technology. In this situation, it is possible to include, as a part of the workflow design, ad hoc semantic translators on each segment of a datapath

between two LDBs.

Using highly specialized translation modules has two advantages: (1) specific knowledge about the pairwise translation rules is localized in each module, and (2) the architecture can be extended incrementally as needed. Although in theory the number of translator required grows exponentially in the number of nodes in the workflow, in practice the number of useful combinations is usually quite manageable.

Integrated Approach. In the fully integrated architecture, all the information about local and global semantics and schema integration is stored and managed in one central site. The main modules in this architecture are a schema manager; a multiquery processor; a global transaction manager; and a colleciton of local translators, one for each LDB in the federation. LDBs are required to join the federation by registering their schema with the schema manager.

One example of this architecture is the Pegasus MDBMS [ASD+91]. In Pegasus, integration can be performed by the users, and it is partially visible by other users through the central site. Semantic heterogeneity is resolved by the schema manager using a semantic dictionary.

This method allows the central site to maintain a single repository for the semantics of the MDB. Centralization also simplifies global concur-

rency control, but at the cost of incresed complexity in the management of multiple semantic translations.

Fully Decentralized Approach. Opposite in nature to full integration is full decentralization (Figure 1.2-(b)), where in the absence of central control, each LDB contains a multidatabase language execution engine and a communication module. Semantic conflict resolution is performed at query formulation time by the multidatabase application developers. The multidatabase language is typically an extension of a standard access language such as SQL, enhanced with features to define multidatabase objects and their correspondence to the local objects. Although integration is normally performed autonomously for each LDB, local sites are free, but not constrained, to define common onthologies to express common database semantics for similar local schemas.

This distributed architecture affords great flexibility in the definition of different, at the expense of a more complex local execution module and more difficult global concurrency control.

Examples of this approach are UniSQL/M and MSQL+ [MR95]. The latter is presented in detail in Section 1.0.7.

Broker-based approach. The general architecture of a broker for the resolution of semantic conflicts [DGH+95] includes a conflict detector mod-

ule that uses shared ontologies and libraries of local contexts (see Figure 1.2-(d)). When the receiver submits a query, the broker generates a conflicts table for the query using shared ontologies and the local context libraries, resolves the conflicts, and then converts the receiver's query into the source context. The source processes the query and sends the result to the broker, which converts the result back to the receiver's context and sends it to the receiver.

The major advantage of this approach is that semantic heterogeneity resolution becomes totally transparent to the user. However, the provess of building shared ontologies and libraries of local contexts has not been completeley automated. Furthermore, if the broker is connected to many sites, and each site has a large number of schemas, then the local context library can grow considerably in size. Finally, some components of the local schema, such as integrity constraints, may be difficult to capture using onthologies.

1.0.4 Summary

We described the definition and the spectrum of semantic heterogeneity in MDBMS. And we discussed the comprehensive classification of semantic heterogeneity. We introduced semantic heterogeneity resolution methodologies developed so far under MDBMS: a simple translation approach, integrated approach, fully decentralized approach, and broker-based approach. We men-

tioned some of the advantages and disadvantages of each method. We believe that a fully decentralized approach with a rich multidatabase language may prove to be a practical way to handle semantic heterogeneity

Functionalities of MDB Query Languages

In this section we outline some historical notes on multidatabase languages, list some basic requirements and design principles, and recall some basic notions of multirelational algebra that will be used in the rest of the chapter.

1.0.5 Historical Perspective

One early proposal for a relational MultiDatabase System is MRDSM, a prototype developed at INRIA as an extension to the MRDS DBMS, based on the Multics Relational Data Store [Lit85]. The main goals of the system were to allow for the retrieval of data from multiple relational sources in a mostly transparent way. The assumption underlying the development of MRDSM and of its successors, mainly MSQL, was to overcome the lack of integration among local schemas by providing an expressive, user-level language that could be used to pose queries to those schemas directly, rather than to a unified global schema. The language, called MML and designed as an extension to SQL, would support the specification of multidatabase queries that include interdatabase joins.

A multiquery processor implements the execution model of MML, by decomposing a MML multiquery into a collection of monodatabase, SQL queries

that can be sent independently to each local SQL processor[6]. The results returned by each query would then be recomposed to form one final relation for the multiquery.

Following the same paradigm, the immediate successor of MRDMS, Multidatabase SQL (MSQL), was designed to provide a rich language for both schema (MultiDDL) and data (MultiDML) manipulation. MSQL is presented in detail in section 1.0.7. The MDDL includes constructs for creating "virtual" multidatabases, copying schema objects definitions from one schema to another, and in general accessing and manipulating multiple Data Dictionaries. The MDML includes such advanced features as dynamic attributes, semantic variables and multidatabase triggers.

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 the corresponding real data items, the multiquery processor can map each variable onto the schemas, yielding a set of elementary queries. The results of these queries can be joined. *Interdatabase joins* actually represent a rudimentary form of data

[6]In particular, a standard SQL query is a MML query directed to a single database.

fusion, through which data retrieved from different sources can be explicitly recombined to yield new information.

A first implementation of MSQL is available as part of the Omnibase project at University of Houston, Department of Computer Science [SRL93]. The prototype, built on top of the Narada multi-application environment [HAB+92], demonstrates the execution of a MSQL SELECT multiquery against several databases residing on heterogeneous platforms and served by different SQL engines. The evaluation plan, compiled into DOL, the scripting language of Narada, includes full handling of distributed, interdatabase joins.

That version of the language was later augmented to include notions of Multidatabase Transactions. Based on the assumptions of local execution autonomy and transaction model heterogeneity, the proposal included using compensating transactions [KLS90] to deal with committed transactions that need to be "semantically rolled back" at sites that do not support the two-phase commit protocol. Databases were also partitioned into the two classes of "vital" and "nonvital" information within the scope of a transaction, depending on the relative importance of a successful commit and the probability of failure at each local site. Transaction semantics was defined to ensure execution correctness with respect to the vital set. These notions, along with support for update multiqueries, was implemented as a second prototype [Mis93b, Bre93].

The experience accumulated with these experiments on the advantages and limitations of MSQL highlighted the need for new features which would address the resolution of schema conflicts, a fundamental aspect of multidatabase processing that the language did not seem to handle adequately. 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, on which standard relational operators can be applied without transformation of the operands. Furthermore, the language offers little help in the case of data type discrepancies and union incompatibility. In general, the expressivity of the language appears to be limited by the lack of external knowledge available to the query processor.

New extensions were proposed and discussed in [Mis93a]. Among these, the explicit inclusion of a *context* surrounding a multiquery and the use of higher-level attributes for conflict resolution.

So augmented, MSQL has become an experiment in "dynamic integration," with increased flexibility being the main benefit over traditional, static integration. Rather than designing a global schema and corresponding translating processors [HML85], a number of global views can be dynamically created and manipulated to reflect the needs of specific multidatabase applications. This version of MSQL, and the corresponding extensions to the query

processor, are described in [MR95] and have been partially implemented as part of the Omnibase project mentioned above.

Finally, we mention the approach to schema integration and query formulation with limited integration visibility, described in [ECR87]. The notions of *connectors* among relations and of *extended abstract data types*, comprehensive representations of knowledge about a domain of data values, 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.

1.0.6 Design Principles

Common to the various experiments in multidatabase access, outlined in the previous section, is a collection of basic requirements that multidatabase languages (MDBL) should satisfy:

multitable manipulation. At the minimum, a MDBL should support simultaneous manipulation of tables in different schemas. The MSQL notion of *multitables* extends individual relational tables to *sets* of ta-

bles that can be referred to using a single identifier;

subsumption. A monodatabase query is a particular case of a multidatabase query. The language should extend a standard DB language so that standard queries are executed according to their usual semantics;

local schema accessibility. As a consequence of the subsumption requirements, the language should allow direct reference to local database objects, as well as to export schema objects that may have been defined for the sake of joining the federation;

location. Location transparency is not enforced, i.e., a global naming scheme may be used (as is the case for commercial distributed DBMSs) to refer to distributed objects[7]

meta-data accessibility. The multiquery processor should have access to relevant data dictionary information from each LDBMS, either directly through queries or by keeping private views of the LDBMS schemas;

query contextualization. A multiquery is interpreted with respect to a *context*, which defines the semantics of the multiquery, much in the same way as the customary LDBMS schema constitutes a context for a regular query. In addition, a context should extend the LDBMS objects namespace by providing new names for *multidatabase objects*. Section 1.0.7

[7] However, an alias mechanism can be used to hide location information in the query.

provides examples of multiDB objects. In MSQL+, the context definition is explicit and available to the user, and contexts can be switched, altered and otherwise adapted to represent different multiquery semantics; and

conflict resolution. Finally, the language should provide enough expressivity to allow for semantic conflict resolution at query definition time.

1.0.7 Multirelational Algebra

Multirelational algebra [GLR91] is an extension of relational algebra. A *multirelation* is a *set* of relations. Multirelational operators apply to multirelations, extending relational operators in a natural way, i.e., for each multirelational operator *op*, there is a corresponding relational algebra expression Exp_{op} such that $op(R) = Exp_{op}(R)$.

The following, basic multirelational operators will be useful in the examples presented in the rest of this section. Let $\mathcal{R} = \{R_1, \ldots, R_n\}$, $\mathcal{S} = \{S_1, \ldots, S_m\}$ be two multirelations, let $Attr(R) = \{R.A_1, \ldots, R.A_k\}$ identify the set of attributes of a relation R, and let p be a predicate on the set of attributes $\bigcup_i Attrs(R_i)$.

Projection

$$\pi_{A_1,\ldots,A_n}(\mathcal{R}) = \{\pi_{A_1,\ldots,A_n}(R_1), \ldots, \pi_{A_1,\ldots,A_n}(R_n)\},$$

where $\{A_1, \ldots, A_n\} \subseteq \bigcup_i Attrs(R_i)$. Notice that some of the attributes

$A_1, \ldots, A_n$ may not be defined for a particular R_i. In this case, $\pi_{A_1,\ldots,A_n}(R_i)$ is the null relation.

Selection

$$\sigma_p(\mathcal{R}) = \{\sigma_p(R_1), \ldots, \sigma_p(R_n)\};$$

if $A_j \notin Attrs(R_i)$ for some A_j appearing in p, then $\sigma_p(R_i)$ is the null relation.

Join

$$\mathcal{R} \bowtie_p \mathcal{S} = \{R_i \bowtie_p S_j \mid i : 1 \ldots n, j : 1 \ldots m\};$$

again, element $R_i \bowtie_p S_j$ is null if p predicates over attributes not in $Attrs(R_i) \cup Attrs(S_j)$.

Union

$$\mathcal{R} \cup \mathcal{S} = \{R_i \cup S_j \mid i : 1 \ldots n, j : 1 \ldots m\}$$

Difference

$$\mathcal{R}/\mathcal{S} = \{R_i \setminus S_j \mid i : 1 \ldots n, j : 1 \ldots m\}$$

Interdatabase joins, a common operation in multidatabase SQL, are expressed as regular semijoins on the relations returned by elementary queries. An example is presented in the next section.

MSQL+ Approach

In this section, we present the part of MSQL language that deals with read-only queries. We introduce three schemas for our examples, and describe the Data Definition (Section 1.0.9) and the Data Manipulation (1.0.10) sub-languages, giving the semantics of multiquery execution and outlining some implementation issues.

1.0.8 Example Database Schemas

The three schemas we will use to support our running example deal with professional computer associations, SIGMOD, AI*IA, and IFIP, that need to keep track of conferences, memberships, paper submissions, etc. The need for targeted advertisement about some new publication, for instance, may motivate the grouping of those databases into a loose federation. By being able to formulate multiqueries that reveal correlations among heterogeneous data (e.g. the same researcher having attended different types of conferences), advertisers can narrow down their marketing target.

The database schema for SIGMOD is defined as follows[8]:

```
People(ID, institution, e_mail, status)

Subscr(sub_ID, series, sub_start_date)
```

[8]Here and below, underlined attributes are part of the primary key.

```
Proc_sales(buyer_ID, conf_name, n_copies, tot_cost)

Conf(conf_ref, edition, POC, location)

Proc_publ(conf_ID, publisher, editor, procs_cost)

Series_publ(series_ID, issue, publisher, editor, spec_topic)
```

Figure 1.0.8 illustrates this schema. Association members are recorded in the PEOPLE table, together with their membership status. Each member may subscribe to any number of SERIES. For each conference, we record number of copies of the Proceedings bought by each member in the PROCEEDINGS table.

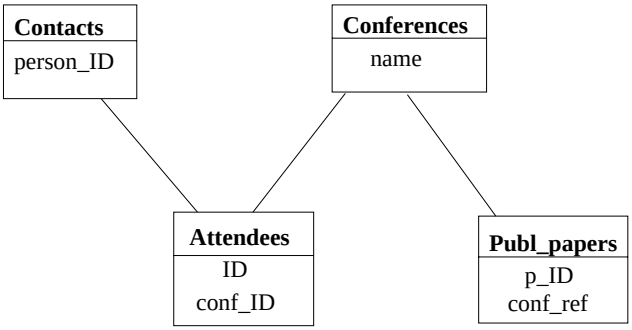


FIGURE 1.3

Schema for the SIGMOD Database.

Here is the database for AI*IA, illustrated in Figure 1.0.8:

```
Contacts(Person_ID, name, member_status, institution, e_mail)
```

```
Conferences(name, type, chaired_by, CFP_text, procs_price)

Attendees(ID, conf_ID, speaker, reg_total, procs_copies)

Publ_papers(p_ID, title, first_author, other_authors,
            keywords, conf_ref)
```

Here members are referred to as CONTACTS, and their attributes are similar, but not the same, as those in SIGMOD. We have conference attendees, rather than general members, who purchase proceedings. Table PUBL_PAPERS records the papers published at each conference.

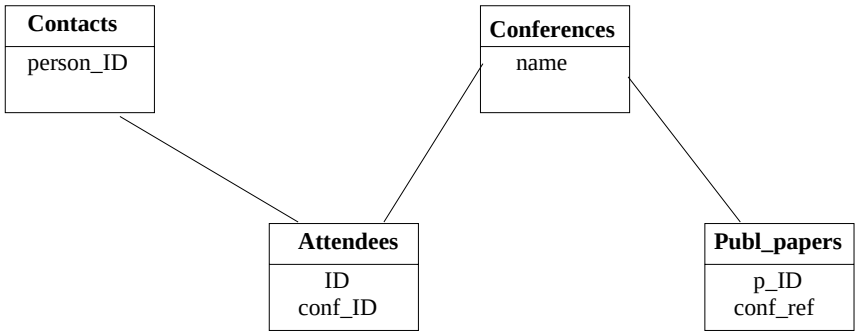


FIGURE 1.4

Schema for the AI*IA Database.

The third database is for IFIP. Being an association of associations, it keeps a table of member societies. Here we have a very simple relationship

linking AUTHORS to PUBL_PAPERS and indirectly, to the conferences table (Figure 1.0.8.

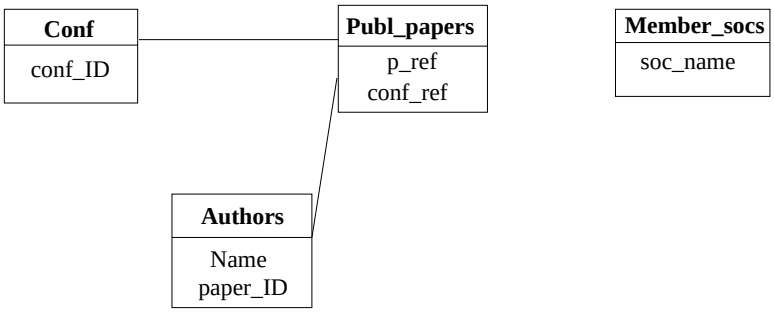


FIGURE 1.5

Schema for the IFIP Database.

```
Member_socs(soc_name, mailbox, start_date, status)

Conf(conf_ID, organized_by, location, procs_publisher,
          publ_date)

Publ_papers(p_ref, title, conf_ref, abstract, full_text_ref)

Authors(Name, e_mail, paper_ID, reviews)
```

1.0.9 Data Definition Language

The extensions to the SQL DDL, proposed in the original MSQL paper [Lit88], reflect a number of assumptions that are crucial to the design of the language itself. First, the federated databases have access to each other's Data

Dictionaries, and each of them may copy fragments of others' schemas into their own. Each schema object creation construct is augmented with a FROM clause to reference corresponding remote schema objects, as in statements for database and table creation:

```
CREATE DATABASE <new database name>
[...]
FROM <source database id>
```

Here the schema defined for <*source database id*> is copied into an identical schema for <*new database name*>. Similarly, but at a finer level of granularity, it is possible to create and alter tables by copying only the definition of single columns:

```
CREATE TABLE <new table name>
            (<column definition> [<column definition>])
FROM <source table id>
```

The usual syntax for <*column definition*> is also augmented to allow for copying from an existing remote column definition:

```
<column definition> ::= [(usual SQL syntax)...]
FROM <source column definition>
```

This syntax suggests a strongly cooperative environment, whereby each local schema results from a variation of a basic, common definition that is agreed

upon by all members. In this scenario, essentially different from that of totally independent schemas evolving in complete autonomy, potential definition conflicts are largely absent, and there is little need for extensive knowledge about local domains. As a consequence, schema reconciliation is hardly ever needed.

In our example databases, on the other hand, we can assume that schemas were designed and continue to evolve independently of each other, yielding a potential for conflicts[9].

The second assumption is that the scope of each statement potentially involving multiple schemas is always made explicit. The scope of a multidatabase statement, be it a query or a DDL command, is defined through the USE clause. In particular, fragments of one schema can be imported into a number of other schemas using a single statement, like the following:

```
USE             DB_A, DB_B
CREATE TABLE    T_common
FROM            DB_Source.original_t;
```

This statement creates two identical copies of table ORIGINAL_T from database DB_SOURCE, in schemas DB_A and DB_B. Also, each schema object can be fully qualified with the schema name, e.g. DB_A.T_COMMON.X, to disambiguate.

[9]In fact, very few of the MSQL import and copy features for object definition can be used to construct those examples.

Finally, the CREATE MULTIDATABASE statement lets a user define a single name for a set of databases, as in:

```
CREATE MULTIDATABASE FED_1 (DB_A, DB_B);
```

Likewise, the ALTER MULTIDATABASE statement allows deletion or insertion from the set of member databases. Using a single identifier offers more than just a shorthand notation for the USE clause. The multidatabase data dictionary, defined as the collection of all schema objects in the scope, can be used to refer to sets of schema objects, e.g. FED_1.T_COMMON now represents the set {DB_A.T_COMMON, DB_B.T_COMMON}.

1.0.10 Data Manipulation Language

We now present some novel database manipulation features that make MSQL suitable for schema and data interoperability. First, we exemplify the use of multiple identifiers and semantic variables, and give the semantics of a multiquery in terms of multirelational algebra (see Section 1.0.7). Semantic variables are collective names defined to refer to collections of local schema objects, and represent a first step towards the definition of an explicit multiquery *context*. By providing the means to achieve some schema abstraction, they help in the resolution of some types of schema conflicts. The use of dynamic attributes, outer join, and outer union are also intended to help in the cases where schemas present some limited form of conflict.

However, as we have pointed out when describing the DDL, one original assumption in MSQL was that these conflicts would occur rarely. Consequently, the ability to handle schema conflicts using these mechanisms alone is naturally limited. In Section 1.0.10, we relax the assumption of non-conflicting schemas. We describe a more general framework which, although requiring the context to be declared explicitly and more exhaustively, allows a better definition of the abstract schema multiqueries operate on. Together with a generalization of semantic variables, we introduce a few additional mechanisms to help in schema conflict resolution, mainly type *coercion*, *user-defined operators*, and *implicit joins completion*.

Multiple Identifiers and Semantic Variables

To begin, let us introduce an elementary multidatabase query that operates on our example databases.

EXAMPLE 1.1

The following multiquery retrieves the names and e_mail addresses of authors who have published a paper both in some IFIP and AI*IA conference, together with the title of the paper:

```
USE       AI_IA, IFIP
SELECT    name, e_mail, title
FROM      Authors, IFIP.Publ_papers IFIP_paper, Contacts,
```

```
            Attendees, AI_IA.Publ_papers
  WHERE     Authors.Name = Contacts.Name
  AND       Contacts.Person_ID = Attendees.ID
  AND       Attendees.speaker = 'Y'
  AND       Authors.paper_ID = IFIP_paper.p_ref;
```

The intended purpose of the query is to retrieve NAME, E_MAIL, and TITLE from both schemas in the scope, by computing some local joins (between CONTACTS and ATTENDEES in AI_IA and between AUTHORS and PUBL_PAPERS in IFIP) and one interdatabase join on the author's name. Identifiers in the SELECT clause are *multiple*: they refer to different attributes in different schemas. The result of this multiquery is the following *set* of two relations known as a *multirelation*:

[Authors.name, Authors.e_mail, Publ_papers.title]

for AI_IA, and

[Contacts.name, Contacts.e_mail, Publ_papers.title]

for IFIP.

The semantics of this multiquery can be expressed using multirelational algebra, as follows:

$$\pi_{name,e\_mail,title}(R_1^{'}, R_2^{'})$$

where

$$R_1 = \text{AUTHORS} \bowtie_{paper\_ID=p\_ref} \text{PUBL\_PAPERS},$$

$$R_2 = \sigma_{attendees.speaker='Y'}(\text{CONTACTS} \bowtie_{person\_ID=ID} \text{ATTENDEES})$$

and

$$R_1^{'} = R_1 \bowtie_{name} R_2,$$

$$R_2^{'} = R_2 \bowtie_{name} R_1$$

Notice that the two semijoins actually express the interdatabase join across the relations produced by each elementary query.

iN particular, when the resulting multirelations are *union-compatible*, the result can be expressed as a single relation whose content is the union of the contents of the relations in the set.

Multiqueries like the one in the example above can only be expressed as long as attribute names are the same (or very similar, if wildcards are allowed in identifiers) across several schemas. This is not the case in general, especially when the federation includes schemas that have been evolving autonomously from each other.

Semantic variables provide a more general naming mechanism whereby different object identifiers can be referred to using a common name.

EXAMPLE 1.2

The following example illustrates the idea. We are interested in advertising a new book on Datalog, and we identify our potential target as people who have an interest both in databases and logic programming. We want to retrieve the addresses of researchers who have in the past bought copies both of SIGMOD and AI*IA Proceedings. The following multiquery uses semantic variables to refer to identifiers in the two schemas:

```
USE        AI_IA, SIGMOD
LET        person.membership BE people.status,
           contacts.member_status
LET        conf.name BE proc_sales.conf_name, attendees.conf_ID
SELECT     person.e_mail, person.membership, conf.name
FROM       person, conf, attendees
WHERE      people.ID = proc_sales.buyer_ID
AND        conf.n_copies > 0
AND        contacts.person_ID = attendees.ID
AND        attendees.procs_copies > 0
AND        people.ID = attendees.ID;
```

In this query, the semantic variables PERSON and MEMBERSHIP, introduced by the LET clauses, are common names for the identifiers on the right-hand

side of the "BE" keyword. The general syntax for the LET clause is:

LET <variable>[.<variable>]... BE <object group> [<object group>]...

<object group> ::= <object name>[.<object name>]...

where the number of elements in each <object group> equals the number of variables on the left-hand side. <object name> can be the name of either a local attribute or a table. In general, variables range over the domains of *table* names and *attribute* names, respectively. We represent a LET clause L using the notation:

$$L = (x_1, \ldots, x_n) \rightarrow \{(A_{11}, \ldots, A_{1n}), \ldots, (A_{m1}, \ldots, A_{mn})\}$$

Upon query evaluation, these variables are pairwise lexically replaced by their respective object identifiers:

$$(\text{PERSON}, \text{MEMBERSHIP}) \rightarrow \{(\text{PEOPLE}, \text{STATUS}), (\text{CONTACTS}, \text{MEMBER\_STATUS})\}$$

and

$$(\text{CONF}, \text{NAME}) \rightarrow \{(\text{PROC\_SALES}, \text{CONF\_NAME}), (\text{ATTENDEES}, \text{CONF\_ID})\}$$

The process of lexical substitution yields the following equivalent set of four multiqueries, without LET clauses:

1.
```
USE        AI_IA, SIGMOD
SELECT     people.e_mail, people.status, proc_sales.conf_name
```

```
FROM        people, proc_sales, attendees
WHERE       people.ID = proc_sales.buyer_ID
AND         proc_sales.n_copies > 0
AND         contacts.person_ID = attendees.ID
AND         attendees.procs_copies > 0
AND         people.ID = attendees.ID;
```

This multiquery is then equivalent to the set of two elementary queries, with no interdatabase joins, obtained by resolving the multiple identifiers. The result is relation R_1 from AI_IA:

$$R_1 = [\text{PEOPLE.E\_MAIL}, \text{PEOPLE.STATUS}, \text{PROC\_SALES.CONF\_NAME}]$$

containing the tuples for which the PEOPLE.ID also meets the conditions in SIGMOD.

2.
```
USE         AI_IA, SIGMOD
SELECT      people.e_mail, people.status, attendees.conf_ID
FROM        people, attendees
WHERE       people.ID = proc_sales.buyer_ID
AND         attendees.n_copies > 0
AND         contacts.person_ID = attendees.ID
AND         attendees.procs_copies > 0
AND         people.ID = attendees.ID;
```

In this query, the object ATTENDEES.N_COPIES resulting from the substitution does not belong to any of the schema in the scope. Therefore, the query is discarded as *non-pertinent* and does not contribute to the final relation.

3.
```
USE         AI_IA, SIGMOD
SELECT      contacts.e_mail, contacts.member_status,
            proc_sales.conf_name
FROM        contacts, proc_sales, attendees
WHERE       people.ID = proc_sales.buyer_ID
AND         proc_sales.n_copies > 0
AND         contacts.person_ID = attendees.ID
AND         attendees.procs_copies > 0
AND         people.ID = attendees.ID;
```

This multiquery results in relation R_2, with attributes from both schemas, namely:

$R_2 = [$CONTACTS.E_MAIL, CONTACTS.MEMBER_STATUS,PROC_SALES.CONF_NAME$]$

where again the tuples satisfy the stated conditions on both schemas.

4.
```
USE         AI_IA, SIGMOD
SELECT      contacts.e_mail, contacts.member_status,
            attendees.conf_ID
```

```
FROM        contacts, attendees
WHERE       people.ID = proc_sales.buyer_ID
AND         attendees.n_copies > 0
AND         contacts.person_ID = attendees.ID
AND         attendees.procs_copies > 0
AND         people.ID = attendees.ID;
```

As for the second query, above, this multiquery is also discarded, since ATTENDEES.PROCS_COPIES is not defined in the scope.

The final result of the original multiquery is multirelation $\{R_1, R_2\}$. If R_1 and R_2 happen to be union-compatible, then the multirelation reduces to the union of the component relations, i.e., $R_1 \cup R_2$.

The semantics of the LET clause can be formalized by introducing a set of operators that capture the decomposition process, as shown in the following example. Given a query Q and a LET clause L, operator

$$Let_{base}(Q, (x_1, \ldots, x_n) \rightarrow \{(A_{11}, \ldots, A_{1n}), \ldots, (A_{m1}, \ldots, A_{mn})\}) = \{Q_1, \ldots, Q_k\}$$

for $k \leq n$ represents the basic decomposition process, producing queries Q_j by lexically replacing each occurrence of x_i in Q with A_{ij}. Notice that, as we have seem in the previous example, $k < n$ since some of the resulting queries may not be *pertinent*, that is, the substitution may yield combinations of table

and attribute identifiers that may be inconsistent for some of the queries.

Next, we define decomposition over a set of queries:

$$Let_1(\{Q_1, \ldots, Q_n\}, L) = \bigcup_{i:1}^{n} Let_{base}(Q_i, L)$$

Decomposition of a set $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$ of queries against a set of LET clauses is defined recursively:

$$Let_{rec}(\mathcal{Q}, \{L_1, \ldots, L_k\}) = Let_{rec}(Let_1(\mathcal{Q}, L_k), \{L_1, \ldots, L_{k-1}\}) \text{ for } k > 0$$

and

$$Let_{rec}(\mathcal{Q}, \emptyset) = \mathcal{Q}$$

Finally, the entire substitution process for multiquery Q and LET clauses $\{L_1, \ldots, L_k\}$ is represented by $Let_{rec}(\{Q\}, \{L_1, \ldots, L_k\})$.

In this example, the use of semantic variables helped create a compact multiquery by overcoming overcoming lexical differences. However, some of the limitations in this approach make it difficult to use:

- the LET-substitution process exhaustively replaces all semantic variables with each legal combination of corresponding local variables. This process yields a number of elementary queries equal to the product of the number of list elements in the right-hand side of each LET clause. Unfortunately, some of the combinations are not meaningful, i.e., a number of spurious queries are potentially generated. Discovering the subset

of meaningful, or *pertinent* queries, requires looking up each identifier in a local data dictionary. Apart from the computational inefficiency, this process makes it hard for the user to figure out the meaning of the multiquery without a careful scrutiny of the database schemas;

- While it seems convenient to "factorize" identifiers in the SELECT clause, figuring out common identifiers for the joining attributes which, once replaced, correctly represent all the intended joins can become a real puzzle. Having to list out explicitly the local and interdatabase joins may defeat the whole purpose of using LET;

- the local identifiers represented by a semantic variable belong to different domains and are not necessarily union-compatible. The LET construct is not expressive enough to account for these differences, as well as to specify how they should be reconciled.

- Semantic variables have an implicit *type* since they may range over different object name domains, e.g. table name and attribute names. Making the type explicit would help understanding their intended meaning better.

Explicit multiquery context and user-defined operators.

We presented semantic variables as a first step towards an explicit representation of the context surrounding the evaluation of a multiquery. The

main limitation of this approach, which is based on the introduction of new identifiers in the name space of database objects, is that these identifiers are simply aliases and are only defined to the extent to which they *refer* to existing database objects.

To solve this problem, [MR95] proposed to introduce a new set of *virtual database objects*, as opposed to just *names*, and to supply a mechanism to relate those virtual objects to existing local objects. In the current proposal, these objects are defined in particular as *attributes* of a new "global" entity, a type of degenerate schema composed of only one entity, that can be referred to in multiqueries. They are typed the same way regular attributes are.

The virtual 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[10]. A *mapping descriptor* maps an expression that has local attributes as operands and uses the defined operators onto a type-compatible virtual attribute.

The main shift from the semantic variables viewpoint is that virtual attributes now exist *independently* of the local schemas. They are generally

[10]This whole design could be rephrased in terms of an object model: the virtual objects become data members of a virtual class, each local schema can be mapped into one class, and functions that are available at local sites become public methods of those classes.

used to describe a particular database domain at a higher level of abstraction than local objects. Virtual objects and local operators are defined in a special *declarations section.*

A multiquery is now evaluated within the context defined by the declarations[11], complemented with the definition of the mapping descriptors required to use the virtual objects.

Virtual objects used in a multiquery are really a particular case of semantic variables where variables are restricted to range only on local attributes, rather than on tables. This is a consequence of grouping together the virtual objects as attributes of a new, higher-level table; since the mappings preserve the type, virtual attributes can only represent type-compatible expressions on local attributes[12]. The trade-off for introducing this restriction is that the resulting context definition is better structured and easier to understand, customize, and reuse. The following example introduces a simple notation for the declaration of virtual objects and illustrates a formulation of the multiquery shown in Section 1.0.10 that uses the new features.

[11] Notice that we now have two notions of scope: the semantic scope, in the MSQL sense, defined as the collection of databases addressed by the query, and the syntactic scope of a declaration with respect to a multiquery. The scope of a declaration may include a whole transaction unit or, hopefully, a whole multidatabase application.

[12] There is no notion, in the current version of the language, of a "higher-level table" that can be mapped onto a local table.

```
Declare
   virtual
            ID: string(50);
            membership:  string(3);
            e_mail:  string(25);
            conference:  string(50);
            copies_sold:  integer;
            amount_for_procs:  real;
   operators
            string(3) status_to_membership(string(1)) @ SIGMOD;
            string(1) membership_to_status(string(3)) @ SIGMOD;
            string(50) to_standard(string25) @ SIGMOD;
            string(3) std_status(short) @ AI_IA;
            string(50) export_format(varchar(30)) @ AI_IA;
   begin
            USE    SIGMOD, AI_IA
            LET    membership BE
                   status_to_membership(people.status),
                   std_status(contacts.member_status);
            LET    e_mail BE people.e_mail, contacts.e_mail;
            LET    conference BE proc_sales.conf_name,
```

```
        attendees.conf_ID;
LET     copies_sold BE proc_sales.n_copies,
        attendees.procs_copies;
SELECT e_mail, membership, conference
WHERE  people.ID = proc_sales.buyer_ID
AND    contact_person_ID = attendees.ID
AND    to_standard(people.ID) =
        export_format(attendees.ID)
AND    copies_sold > 0;
```

end;

The example contains three parts. The first part is the declaration of virtual objects and operators. Objects have a type and a domain. To be useful, the type system for virtual objects must contain the type system assumed for the local schemas, and the type of virtual objects should be more general than that of any local object that can be made to correspond to them through the mapping descriptors.

In the current syntax, the domain of a virtual object is not expressed explicitly[13]. In this example, we assume the domain of MEMBERSHIP is the

[13]In some database systems, domains are expressed separately as constraints on the attributes. In the current version of MSQL, the constraint mechanism and its corresponding sub-language are not yet available.

set {ST, JR, SR, HO}[14].

Operators are declared through their their signature. The local schema where the operator is available is also given, together with additional information, not shown here, on system-dependent details such as the invocation mechanism and the communication protocol. These declarations can be thought of as a collection of export interfaces made available by each local system to the federation. They suggest that the local administrators are responsible for their maintenance. In the example, the SIGMOD site has two functions, one inverse of the other, to translate between the local format for association membership status and the "standard" one. For instance, SIGMOD may use codes such as "S" for "student", "R" for "regular" and "H" for "honorary". Function STATUS_TO_MEMBERSHIP then maps "S" into "ST", "R" into "JR", and so forth. AI_IA only defines one function for its numeric codes for the status, to the "standard" one. In addition, the two functions TO_STANDARD and EXPORT_FORMAT change the format of subscriber identifiers in the respective schemas. These functions are used to reconcile mismatched ID attributes in the query interdatabase join.

In the second part of the example, LET clauses introduce the mappings. The new syntax for the LET clause is simply:

LET (<virtual object> [,<virtual object>, ...]) BE

[14]For STudent, JunioR, SenioR and Honorary, respectively.

(<local expression> [, <local expression>],...) [,

(<local expression> [, <local expression>],...)]

] ...

where each <local expression> is an expression on local, fully qualified attributes that may involve local operators declared as above, as well as local built-in database functions[15]. Notice that all virtual objects now range over the same domain (expressions on local attributes), and, unlike for the regular LET, it is now possible to perform type-checking on the mappings.

The notion of mapping descriptors has been formalized to some extent in [MR95], by introducing a functional MD as follows.

Let $\mathcal{S}$ be the the multiquery scope, and $\mathcal{VO}$ the set of virtual objects. For each schema $s \in \mathcal{S}$, let $\mathcal{A}_s$ be the set of all local (table qualified) attributes for database s. Also, let $\mathcal{F}$= $\{f_i : \tau_1, \ldots, \tau_{n_i} \rightarrow \tau\}$ represent the collection of signatures for the declared operators. For a given pair $s \in \mathcal{S}$, $vo \in \mathcal{VO}$ the *Mapping Descriptor*:

$$MD^s_{vo} =< A, f, f' >$$

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

[15] The requirement that each operator in the expression be local can actually be relaxed, if the implementation supports the execution of remote functions on local arguments.

some *vo* and *s*.

Using this notation, the MDs for the example can be written as follows (I indicates the identity function, S is a short for SIGMOD and A for AI_IA):

$$
\begin{aligned}
MD^{S}_{\text{MEMBERSHIP}} &= < \{\text{PEOPLE.STATUS}\}, \text{status\_to\_membership}, \\
&\quad \text{membership\_to\_status} > \\
MD^{A}_{\text{MEMBERSHIP}} &= < \{\text{CONTACTS.MEMBER\_STATUS}\}, \text{std\_status}, \perp > \\
MD^{S}_{\text{ID}} &= < \{\text{PEOPLE.ID}\}, \text{to\_standard}, \perp > \\
MD^{A}_{\text{ID}} &= < \{\text{CONTACTS.PERSON\_ID}\}, \text{export\_format}, \perp > \\
MD^{S}_{\text{E\_MAIL}} &= < \{\text{PEOPLE.E\_MAIL}\}, I, I > \\
MD^{A}_{\text{E\_MAIL}} &= < \{\text{CONTACTS.E\_MAIL}\}, I, I > \\
MD^{S}_{\text{CONFERENCE}} &= < \{\text{PROC\_SALES.CONFERENCE\_NAME}\}, I, I > \\
MD^{A}_{\text{CONFERENCE}} &= < \{\text{ATTENDEES.CONF\_ID}\}, I, I > \\
MD^{S}_{\text{COPIES\_SOLD}} &= < \{\text{PROC\_SALES.N\_COPIES}\}, I, I > \\
MD^{A}_{\text{COPIES\_SOLD}} &= < \{\text{ATTENDEES.PROC\_COPIES}\}, I, I >
\end{aligned}
$$

In the third part of the example, the multiquery refers to the declared virtual objects and operators, both in the SELECT clause and in the interdatabase join. The new LET-substitution process is based on the observation that the mappings are now defined separately between each local schema and the vir-

tual objects, making it possible to carry out the substitutions by considering one schema at a time. The resulting algorithm is somewhat orthogonal to the one outlined in section 1.0.10, where the semantic variables are exhaustively and blindly replaced in the multiquery with each legal combination of local identifiers. For each database in the scope, the virtual objects are replaced by the corresponding expressions in the mappings defined for that database. This allows, for example, the detection of a missing attribute (the mappings are partial) in the SELECT clause and the appropriate insertion of a NULL placeholder. In the process, the set of joins and selections in the WHERE clause is partitioned, yielding one query for each schema, plus one query for the interdatabase joins. The result is a set of elementary queries, one for each database, each with all of and only its own joins. In each query, the SELECT clause is augmented to include the attributes required to perform the final interdatabase join[16]. Notice also that the FROM clause can be omitted because it can be inferred from the use of the local attributes after substitution of the virtual objects.

The new substitution process yields the following two elementary queries (each still containing one interdatabase joins):

```
Elementary Query EQ1 for database SIGMOD:
```

[16] See [Sua92] and Section 1.0.11 for the derivation of a basic evaluation plan and execution of MSQL.

```
SELECT     people.e_mail, status_to_membership(people.status),
           proc_sales.conf_name, to_standard(people_ID)
FROM       people, proc_sales
WHERE      people.ID = proc_sales.buyer_ID
AND        proc_sales.n_copies > 0;
```

```
 Elementary Query EQ2 for database AI_IA:
SELECT     contacts.e_mail, std_status(contacts.member_status),
           attendees.conf_ID, export_format(attendees.ID)
FROM       contacts, attendees
WHERE      contacts.person_ID = attendees.ID
AND        attendees.procs_copies > 0;
```

This version of the example highlights some important differences as compared to the original version of MSQL. First, this controlled use of LET ensures that the relations resulting from each elementary query are all type-compatible. As a result, relational algebra is sufficient to define the semantics of the language, i.e., multirelational algebra need not be used. Then, some schema and data conflicts among mismatched attributes can be resolved using the mapping descriptors involving ad hoc operators in the expressions [MR95].

In Section 1.0.11, we present an extension to the basic evaluation plan for MSQL [Sua92], which takes into account the invocation of the remote func-

tions that implement the external user-defined operators. The multidatabase system architecture must be redesigned accordingly.

To conclude this overview off the MSQL Data Manipulation Language, we mention two additional features, namely *type coercion* and *implicit joins completion*, that help solve union compatibility, schema isomorphism, and key mismatch problems.

Type compatibility

Type compatibility among equivalent, or corresponding, attributes is a necessary step to achieve union compatibility among relations. Since each local DBMS has its own type system, and virtual database objects are meant to represent local database objects, the type system defined for the multidatabase should include all the types defined by the local DBMS, and introduce new, more general types for the virtual objects. The resulting type lattice allows the processor to perform type coercion automatically when needed; this is done by checking type compatibility at the multidatabase level. In the example of Fig.1.0.10, if local attributes PROC_SALES.N_COPIES and ATTENDEES.PROCS_COPIES have different types, say SHORT and NUMBER, respectively, then the type of COPIES_SOLD should be more general than both of them. Upon decomposition, the processor should insert type conversion routines in the evaluation plan, either to transform one of the two types into the other, or to transform each into their join type.

Implicit joins completion

In the multiquery of example 1.0.10, local as well as interdatabase joins are listed explicitly. In some cases, given enough information to the (multi)query processor, some or all of the local joins can be left implicit, i.e., they can be omitted from the query. The notion of *implicit joins* was first introduced and formalized in the study of Universal Relation Databases [MU83]: given a set of joins, or "natural dependencies" on a set of relations, defined independently of a specific query and involving a set A of attributes –the set of attributes in a query projection, there exists an algorithm to determine, for each subset of A, a subset of the given joins that connect those attributes and is minimal with respect to some metric.[17] A query in which some of the local joins are left implicit is called *incomplete*. In practice, referential integrity constraints, often defined with the schema, represent the basic natural dependencies.

[17] Formally, join completion relies on the notion of *join dependency*: given n sets of attributes $R_1, \ldots R_n$, the join dependency $\bowtie (R_1, \ldots, R_n)$ is satisfied by a relation r over $R = \bigcup_{i=1}^{n} R_i$ if and only if: $r = \bowtie_{i=1}^{n} \pi_{R_i}(r)$.

An *object* is defined as a minimal sets of attributes $R_i = \{A_{j_1}, \ldots, A_{j_i}\}$ such that the join dependency $\bowtie (R_1, \ldots R_n)$ holds.

A join dependency is represented as a hypergraph[18], with one node for each attribute appearing in one or more of the R_is, and one edge for each R_i, consisting of all its member attributes.

The algorithm presented in [MU83] finds the minimal lossless joins connecting a subset of attributes in the universal relation, when and only when the hypergraph is acyclic.

The idea of using the implicit joins completion algorithm has been applied by Litwin [Lit85] to the multidatabase environment. If, in fact, natural dependency information is available for each schema, then an incomplete multiquery can be first decomposed into a set of elementary, incomplete queries (using the algorithm described in Section 1.0.10), and then the completion algorithm can be applied to each such queries, *separately*. The key here is on the separation of contexts during completion. If the schemas addressed by the query are not isomorphic. If the navigation required to connect corresponding attributes through joins differs from schema to schema, then by omitting the local joins altogether we gain in expressivity when formulating the initial multiquery, since we can now design an *more abstract* query that hides all those local navigation details. Furthermore, since the decomposition algorithm allows each elementary query to be evaluated within its own context independently of each other, those joins can be completed independently. The trade-off for this convenience is the extra information about the dependencies in each local schemas that is required by the completion algorithm.

This technique, although potentially very powerful, presents the well-known, substantial limitation that the set of minimal completions may not be unique. Intuitively, dependency information may not be sufficient to determine a unique *meaning* for each subset of the attributes and for arbitrary selections. In [MU83], the notion of maximal objects was introduced to deal

with this potential ambiguity. The completion algorithm defined in [Lit85] takes the union of the completed queries resulting from each completion for a given schema.

Since none of these approaches is entirely satisfactory with respect to the way the original, intended meaning of a query is reconstructed, in the current version of MSQL, no predefined strategy is enforced to resolve the ambiguities. Rather, if the query admits more than one completion, the available choices are reported to the user.

The input to the completion algorithm is composed of two parts, the Database Graph (DBG), which includes the *natural dependencies* and forms part of the context (independent of query instances), and the Query Graph (QG), which represents the local joins that are explicitly included in a specific query. In our running example, the context would include a section for the declaration of the dependencies, as follows[19]:

```
For SIGMOD:
    people.ID = proc_sales.buyer_ID
    people.ID = subscr(sub_ID)
    conf.conf_ref = proc_sales.conf_name
    proc_publ.conf_ID, conf.conf_ref
    series_publ.series_ID = subscr.series
```

[19] In this case, we actually list some of the referential integrity constraints.

```
For AI_IA:

contact_person_ID = attendees.ID

conferences.name = attendees.conf_ID

publ_papers.conf_ref = conferences.name
```

DBG nodes represent database relations, while arcs represent relational operators. In our simple example, only equality operators are used. In the general definition, an arc $\{< R.A_i, op_i, S.B_i >\}$ for $1 \leq i \leq n$, represents the predicate $\bigwedge_{i=1}^{n}(R.A_i < op_i > S.B_i)$, while an arc 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 θ-join between relations R and S.

Using these declarations, local joins can be omitted from the multiquery 1.0.10, yielding the following, simpler query:

```
USE     SIGMOD, AI_IA
LET     membership BE
            status_to_membership(people.status),
            std_status(contacts.member_status);
LET     e_mail BE
            people.e_mail, contacts.e_mail;
LET     conference BE
```

```
            proc_sales.conf_name, attendees.conf_ID;
LET     copies_sold BE
            proc_sales.n_copies, attendees.procs_copies;
SELECT  e_mail, membership, conference
WHERE   to_standard(people.ID) = export_format(attendees.ID)
```

The QG is constructed by the processor from the local joins in the query, as follows (in our example, the QG is empty for both elementary queries derived from the simplified multiquery above):

- the QG has one node for each relation used (in the FROM clause) in the query; thus, the set of QG nodes is a subset of the DBG nodes;
- the QG has an arc labeled $<A, B, relop>$ between nodes R and S, for each join $R.A$ *relop* $S.B$ in the query, and for each *theta*-join:

$$r(R.A_1, \ldots, R.A_k, S.B_1, \ldots, S.B_h)$$

The completion algorithm, described in more detail in [Lit85] and implemented as described in [Mis93b], finds the set of minimal-depth spanning trees for the union of the DBG and the QG, giving priority to the QG and adding arcs as necessary to generate a connected component. Note that if, in particular, the QG is already connected (enough local joins were explicit), then the query is considered complete.

In our example, the nodes to be connected are PEOPLE, PROC_SALES and CONTACTS, ATTENDEES for SIGMOD and AI_IA, respectively (see Figg. 1.0.8 and 1.0.8). In both these very simple cases, the algorithm finds a direct arc to connect the two tables and terminates by returning the two complete queries (with inclusion of the interdatabase joins that did not take part in the completion process) without ambiguities; the simplified multiquery has been decomposed and successfully reconstructed for the local schema.

Outer joins and outer union

We conclude our discussion on the language features by observing that user-defined operators, introduced in 1.0.10, effectively generalize both the outer join and the outer union operators. In fact, outer joins are a particular case of θ-joins, where the θ operators may be user-defined. However, sometimes the full power of external operators may not be needed, and outer joins, available in several commercial versions of SQL, can be particularly useful in the multidatabase context. A common situation occurs when trying to join on sets of columns that would logically have a primary-foreign key relationship. In the monodatabase case, referential constraints can often be enforced so that these joins are guaranteed to return all relevant tuples. In the multidatabase case, however, these constraints are generally not available, leading to the case of "children" in one table without corresponding "parents "in another schema. Outer joins resolve this problem.

The outer union operator can be used to compute the union of partially compatible relations by simply returning the union of the two sets of attributes in the two relations. Of course, this operator does not consider the semantics of the attributes involved. Since by keeping all attributes, we only look at their *name*, this operation may result in sets of differently named attributes with similar meaning in the resulting relation.

In our examples schemas SIGMOD and AI_IA, suppose that we want to retrieve the union of [PEOPLE.ID, PEOPLE.STATUS] and [CONTACTS.PERSON_ID, MEMBER_STATUS]. Using an outer union operator, we would have:

```
SELECT       ID, status
FROM         people
OUTER_UNION
SELECT       person, member_status
FROM         contacts
```

This results in relation:

[PEOPLE.ID, PEOPLE.STATUS, CONTACTS.PERSON_ID, MEMBER_STATUS]

If the pairs of attributes PEOPLE.ID, CONTACTS.PERSON_ID and PEOPLE.STATUS, MEMBER_STATUS are not union-compatible, then a user-defined operator that looks at each instance of the attributes may be more appropriate than outer union.

1.0.11 Multiquery Evaluation

The first available implementation of MSQL, described in [Sua92], followed the decomposition process outlined in Section 1.0.10 and does not include external operators. A later implementation [MR95] followed the semantics described in Section 1.0.10, including steps to evaluate external operators. The evaluation strategy for MSQL presented in this section is concerned only with the coordination of the execution of the elementary queries produced by the decomposition phase, regardless of which decomposition semantics is adopted. Thus, we first present a common basic evaluation plan and then detail the additional steps required to invoke user-defined operators that are available as local services or applications.

In the available prototype, the evaluation plan is executed in the Narada multi-system application execution environment [HAB+92]. Narada provides basic multi-platform communication services and specialized connectivity to local services, in particular to local DBMSs. The scripting language DOL, for Distributed Operation Language, is available to program the control and data flow for a Narada multi-system application. DOL provides constructs to define elementary tasks that are to be executed on the remote systems, to express control and data dependencies among said tasks, to specify their parallel execution and synchronization points, and to evaluate conditions based on the status information the tasks return upon completion.

Basic evaluation for SELECT

The basic MSQL processor consists of a query analyzer and decomposer, which produce the evaluation plan, and of a back-end code generator that compiles the plan into a DOL script. The design of the interface between these two modules allows the processor to be targeted to a different object language by replacing the back-end. The basic plan illustrated in Fig. 1.0.11 refers to the elementary queries EQ1 and EQ2 in Section 1.0.10. It assumes that local systems do not provide facilities to execute interdatabase joins[20].

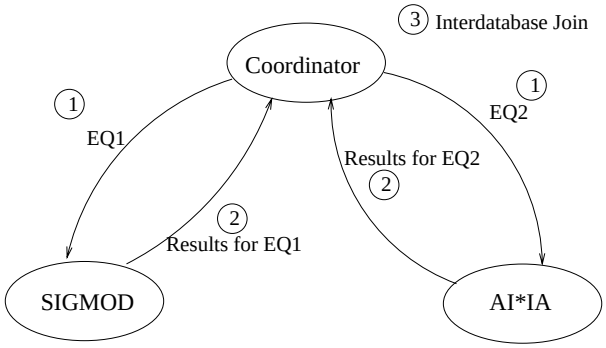


FIGURE 1.6

Basic steps in Multiquery Evaluation

The multiquery is issued and decomposed at the *coordinator site* where

[20] If they do, then the problem reduces to executing a distributed query using a Distributed Database System.

we want the final answer to be delivered. SQL queries EQ1 and EQ2 are sent in parallel to the SIGMOD and AI_IA DBMSs sites, respectively[21]. In the actual architecture, a request for query execution is sent to the two Local Access Manager (LAM) modules that act as proxy users of the local DBMSs on behalf of the coordinator and handle the local database access. Each LAM submits the query, receives the results and returns both the results, and the status condition to the coordinator. The coordinator waits until both results are returned. If both queries are executed successfully, their results are used to perform the final interdatabase join, and the final result is produced.

The actual evaluation plan must take into account the different resources available at each local site. For instance, if the coordinator does not itself have access to a local DBMS, the final join can either be carried out locally at the shell level (operating on files), or the join can be performed at one of the local sites. In the latter case, temporary tables must be setup on the designated database, and individual query results must be re-routed appropriately.

Multiquery optimization, investigated for instance in [Mon93], is still largely an open issue, complicated by the diversity of local resources that must be taken into account.

[21] Remember that these queries return the additional attributes required to carry out the final interdatabase join.

Extended Evaluation with external operators

We now concentrate on the evaluation of a multiquery in the presence of external operators. Two basic types of operators 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

$$\ldots \text{WHERE} \ldots p(R.A_{i_1}, \ldots, R.A_{i_n})$$

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.

To illustrate this concept, and for simplicity's sake, we assume 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. In a more general distributed architecture scenario, local applications would be made available through some registration and brokerage environment, such as DCE, in some uniform way. We further assume, without loss of generality,

that operators and its operands are available at the same site[22].

We indicate the conversion to and from the textual representation by the functions $db\_to\_file(SQL\_query, file)$ and $file\_to\_db(file, DB\_table)$. Figure 1.7 illustrates the main problem with the application of filters and predicates, namely that the input relations must be "broken down" to accommodate the input format accepted by the application and the output relations must recomposed upon completion to be reinserted into the database. A filter f(A,B) applied to table R(ABC) requires first the conversion 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

[22]If this is not the case, additional steps must be inserted in the evaluation plan to migrate the operands, which are usually files containing the relations.

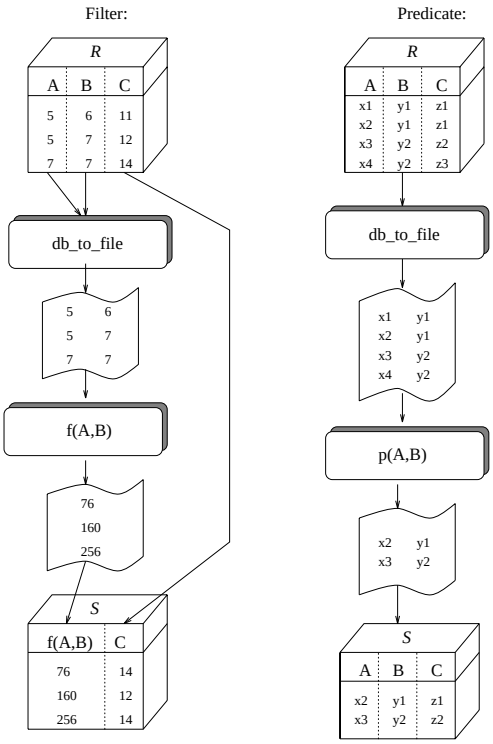


FIGURE 1.7

Effect of filter and predicate evaluation

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 are more opportunities for optimization of the resulting task sequence.

Updates in Multidatabase Languages

In this section, we consider updates in the context of Multidatabase Languages. We introduce the extensions to MSQL proposed in [SRL93] for Multidatabase Transactions, outline the basic evaluation plan for MSQL updates, and describe the UNISQL approach to handling updates.

1.0.12 Updates in MSQL and Multitransactions.

In section 1.0.11, we presented MSQL language constructs for read-only multiqueries, that extend the SQL SELECT query construct to address a number of databases, possibly heterogeneous. Likewise, MSQL extends write operations (UPDATE, INSERT, DELETE) to modify the state of *multiple* databases simultaneously. When the subqueries in a multiple update operation are related to each other, it should be possible to execute them as if they were part of a single transaction.

EXAMPLE 1.3

We would like to increase the cost of the proceedings book for a given conference by 10

```
USE SIGMOD, AI_IA
LET (conf, price) BE    (proc_publ.conf_ID, proc_publ.procs_cost),
                        (conferences.name, conferences.procs_price)
```

```
UPDATE   proc_publ, conf
SET      price = price * 1.1
WHERE    conf = :conf_name
```

where the notation :<variable> indicates an input value for the user. Notice also that, unlike in SQL, here the UPDATE keyword can be followed by a *list* of all the tables that are to be updated. However, each table must belong to a different database, so that upon decomposition, each resulting monodatabase update contains exactly one table in its UPDATE clause.

Decomposing this multiquery yields two elementary, monodatabase queries that could be submitted independently. However, we would like to enforce a transactional semantics on their execution, so that either they both succeed, or none of them has any effect. At the system level, however, encapsulating the two queries in a single transaction may not be feasible, due to the difference in local transaction protocols and the local autonomy requirements. Suppose, for instance, that the DBMS for AI_IA does not support the 2PC protocol, i.e., it works only in autocommit mode. After the two updates have been submitted, at the global level, we are left with very little control on the local AI_IA transaction. In fact, if SIGMOD unilaterally aborts its transaction after AI_IA has already committed, we cannot simply rollback the AI_IA transaction, and the global state for the book price is inconsistent. In this case, at the global level, failure atomicity cannot be enforced.

The capability to control global execution while respecting local autonomy requires both a non-traditional notion of global consistency and the use of extended transaction models [GMS87, KR88, Elm92].

In this section, we present the language features that have been proposed [SRL92] to incorporate some elements of *Flexible Transactions* [Elm92, SANR92], notably *vital databases* and *compensating transactions*.

Vital databases allow the user to specify the desired level of consistency for the execution of a particular multiple update. Since, due to the different types of transaction control available at the local systems, global atomicity cannot always be attained, users may specify a subset of *vital* queries that must be executed together (i.e., they must be either committed or aborted atomically) in order for the global update to be consistent. Let us assume, in our previous example, that the SIGMOD database can be updated using 2PC, while AI_IA works in autocommit mode. Then we can express the fact that SIGMOD is a vital database by extending the USE clause as follows:

```
USE SIGMOD VITAL, AI_IA
```

Assuming, in general, that a multiple query is decomposed in such a way that, at most, one local query is generated for each database, then the VITAL designators attached to the databases are effectively related to the local queries. In this case, the execution of a multiple query is successful when all vital

queries are committed; it aborts when all vital queries are rolled back and is considered incorrect when some of the vital queries are committed and others are rolled back. The final state of all nonvital queries is immaterial to the final state of the global query. In other words, failure atomicity of the global query is enforced only with respect to the vital set.

The vital set for a multiquery should be defined in accordance with the different transaction protocols available locally. If all vital databases support 2PC, then the commit point of all corresponding subtransactions can be synchronized, and failure atomicity can be enforced. Notice that the same property holds true in other particular cases. For instance, the example above is a special instance of a *saga* [GMS87] where there is only one non-2PC transaction, the one for AI_IA. In this case, the SIGMOD transaction can be held in *prepared-to-commit* state until the final state for the AI_IA is known and then be committed or rolled back according to that state.

The semantics of vital designators is not applicable when databases that do not support 2PC are included in the vital set, since, in general, failure atomicity cannot be enforced if the *prepared-to-commit* state is not visible. The notion of *compensation* [KLS90, GMS87] has been adopted in MSQL to deal with this problem. Compensating actions are queries that attempt to return a database to a consistent state following an unwanted commit, by "semantically undoing" the effect of a transaction. Notice that a semantic

undo is not the same as a transaction rollback. In fact, since the original transaction is already committed, the isolation property does not hold since the database state between the commit and the compensation is visible to other transactions. As a consequence, the state after compensation is *not*, in general, the same as the state that existed prior to the offending transaction.

The complete notation for a MSQL multidatabase transaction with compensating action is best illustrated using our familiar example:

```
BEGIN MULTIRANSACTION
USE SIGMOD, AI_IA
LET (conf, price) BE    (proc_publ.conf_ID, proc_publ.procs_cost),
                        (conferences.name, conferences.procs_price)
UPDATE   proc_publ, conferences
SET      price = price * 1.1
WHERE conf = :conf_name

COMP AI_IA
UPDATE   conferences
SET      procs_price = procs_price  1.1
WHERE    name = :conf_name
END MULTITRANSACTION
```

The COMP clause introduces the compensating action for the AI_IA database, which amounts to revoking the 10% price increase produced by the initial transaction. As noted above, since the state of the IA_IA database is exposed between the transaction commit and its compensation, the effect of the compensation is not guaranteed to return the conference price to its original value–it simply divides the *current* price value, whatever it is, by 10%.

Execution model for UPDATE

In Section 1.0.11, we presented the basic execution plans for a read-only multiquery. A similar procedure is applied for updates, with the main difference being the coordination of local transactions and the execution of compensating actions when necessary.

The global update in the previous example yields the three following monodatabase update queries:

For SIGMOD (Eq1):

```
UPDATE proc_publ
SET    procs_cost = procs_cost * 1.1
WHERE proc_publ.conf_ID = :conf_name
```

For AI_IA (Eq2):

```
UPDATE conferences
SET    procs_price = procs_price* 1.1
```

```
WHERE conferences.name = :conf_name
```

Compensation for AI_IA (Eq2.comp):

```
UPDATE  conferences
SET     procs_price = procs_price  1.1
WHERE name = :conf_name
```

The basic evaluation plan is illustrated in Fig.1.0.12. First, queries EQ1 and EQ2 are generated through decomposition at the coordinator site and submitted to the local systems. Similar to the procedure for SELECT, the request for query execution is actually sent to the two Local Access Manager (LAM) modules that act as proxy users of the local DBMSs on behalf of the coordinator and handle the local Database access. For SIGMOD, the 2PC protocol is followed. SIGMOD is expected to return a notification of its pre-commit state if the query is successful. AI_IA, on the other hand, will only notify of its commit or abort condition. This information is then evaluated by the coordinator. If SIGMOD is in pre-commit, then if AI_IA has committed a request for commit is sent to SIGMOD, otherwise SIGMOD is rolled back. In either case, no compensation is necessary. However, if SIGMOD has aborted its transaction while AI_IA has committed, then the compensating action

EQ2.COMP is submitted to AI_IA.

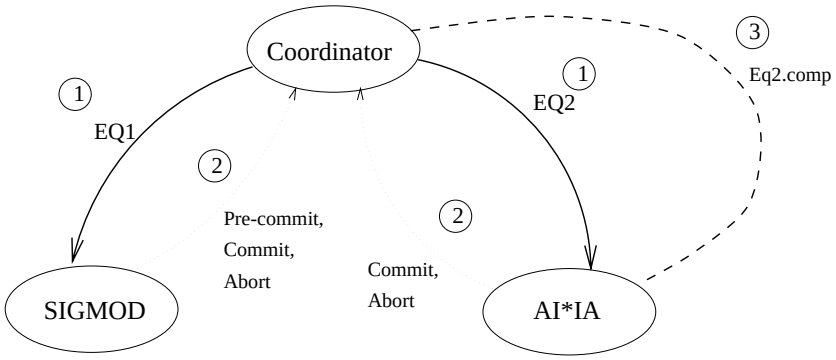


FIGURE 1.8

Transactions with compensation for MSQL updates

An extension of the MSQL processor to produce this simple evaluation plan, together with a DOL implementation of the plan, are described in [Bre93] and [Mis93b]. The implementation also handles interdatabase joins in multiple updates. Similarly to the case of SELECT, to perform interdatabase joins a number of temporary tables must be created, populated, and finally dropped at some local sites or the coordinator site. These housekeeping operations must be part of the global transaction for the multiquery.

Multiple UPDATEs with user-defined operators

To conclude this section, we notice that multiple updates that use expressions in their LET clauses (see Section 1.0.10) are subject to some restriction. In particular, the operators that appear in the expressions on local database objects must be invertible (Section 1.0.10) for the decomposition to work. To illustrate this point, consider the following example, where we would like to change the membership status for some members in SIGMOD and AI_IA:

```
BEGIN MULTIRANSACTION
USE SIGMOD, AI_IA
LET (ms) BE (status_to_membership(people.status),
             std_status(contacts.member_status));
UPDATE   people, contacts
SET      ms = :new_status
WHERE ...
END MULTITRANSACTION
```

The elementary update for SIGMOD is:

```
UPDATE   people
SET      status = membership_to_status(:new_status)
WHERE ...
```

where MEMBERSHIP_TO_STATUS is the inverse function of STATUS_TO_MEMBERSHIP, as defined in the mapping descriptor introduced in Section 1.0.10. However, we can see from the mapping descriptor for CONTACTS.MEMBER_STATUS that operator STD_STATUS has no inverse. Therefore, in this case the elementary update for AI_IA cannot be defined.

The only way LET-substitution can be used in these cases is by supplying inverse operators whenever update operations on the corresponding local database objects are envisioned.

Bibliography

[ASD+91] R. Ahmed, P. De Smedt, W. Du, W. Kent, et al. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24(12):19–27, Dec. 1991.

[Bre93] M. Bregolin. Extensions of MSQL: notes on an implementation. Internal Report, University of Houston, Dept. of Computer Science, March 1993.

[Car94] M.J. Carey et al. Towards heterogeneous multimedia information systems: The garlic approach. Technical Report Technical Report RJ9911.

[DeM89] L. DeMichiel. 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, February 1989.

[DGH+95] A. Daruwala, C. Goh, S. Hofmeister, K Hussein, S. Madnick, and M. Siegel. Context interchange network prototype. In *Proceedings of Sixth IFIP International Conference on Database Semantics*, Atlanta, 1995.

[ECR87] D. Embley, B. Czejdo, and M. Rusinkiewicz. An approach to schema integration and query formulation in federated database systems. In *Proceedings of the Third International Conference on Data Engineering*, February 1987.

[Elm92] A. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan-Kaufmann, February 1992.

[EPS93a] E.-P.Lim and J. Srivastava. Attribute value conflict in database integration: An evidential reasoning approach. Technical Report TR 93-14, University of Minnesota, Dept. of Computer Science, February 1993.

[EPS+93b] E.-P.Lim, J. Srivastava, et al. 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.

[GLR91] J. Grant, W. Litwin, N. Roussopoulos, and T. Sellis. An algebra and calculus for relational multidatabase systems. *IEEE-IMS 1991, Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, April 1991.

[GMS] C. Goh, S. Madnick, and M. Siegel. Context interchange: Overcoming the challenges of large-scale interoperable database systems in a dynamic environment. In *Proceedings of the Third International Conference on Information and Knowledge Management.*

[GMS87] H. Garcia-Molina and K. Salem. SAGAS. In *Proceedings of ACM SIGMOD Conference*, pages 249–259, San Francisco, California, May 1987.

[Gru] Thomas R. Gruber. The role of common ontology in achieving sharable, reusable knowledge bases.

[HAB+92] Y. Halabi, M. Ansari, R. Batra, W. Jin, G. Karabatis, P. Krychniak, M. Rusinkiewicz, and L. Suardi. Narada: An Environment for Specification and Execution of Multi-System Applications. In *Proceedings of the Second International Conference on Systems Integration*, 1992.

[HC96] G. Hammilton and R. Cattel. *JDBC : A Java SQL API*, 1996.

[HM93] J. Hammer and D. McLeod. 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, January 1993.

[HML85] D. Heimbigner and D. McLeod. A federated architecture for information management. *Proceedings of the IEEE International Conference on Data Engineering*, 3(3), July 1985.

[Hu95] X. Hu. Making multidatabases active. Master's thesis, University of Houston, May 1995.

[JNRS93] W. Jin, L. Ness, M. Rusinkiewicz, and A. Sheth. Concurrency control and recovery of multidatabase workflows in telecommunication applications. In *Proceedings of ACM SIGMOD Conference*, pages 456–459, Washington, DC, May 1993.

[KCGS92] W. Kim, I. Choi, S. Gala, and M. Scheevel. On resolving schematic heterogeneity in multidatabase. Technical report, UniSQL, Inc., Austin, TX, August 1992.

[KGK+95] W. Kelley, S. Gala, W. Kim, T. Reyes, and B Graham. *Schema Architecture of the UniSQL/M Multidatabase System*, pages 621–648. ACM Press, 1995.

[KLK91] R. Krishnamurthy, W. Litwin, and W. Kent. 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, May 1991.

[KLS90] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference on VLDB*, 1990.

[KR88] J. Klein and A. Reuter. Migrating transactions. In *Future Trends in Distributed Computing Systems in the 90's*, Hong Kong, 1988.

[KS91] W Kim and J. Seo. Classifying schematic and data heterogeneity in multidatabase systems. *IEEE Computer*, 24(12):12–18, Dec. 1991.

[KSS95] V. Kashyap, K. Shah, and A. Sheth. Metadata for building the multimedia patch quilt. In S. Jajodia and Eds. V.S.Subrahmaniun, editors, *Multimedia Database Systems: Issues and Research Directions*. Springer-Verlag, 1995.

[L+90] W. Litwin et al. MSQL: A Multidatabase Language. *Information Sciences*, 49(1-3):59–101, October-December 1990.

[Lit85] W. Litwin. Implicit joins in the multidatabase system MRDSM. In *Procs. IEEE-COMPSAC*, Chicago, Oct. 1985.

[Lit88] W. Litwin. From database systems to multidatabase systems: Why and how. In *British National Conference on Databases.* Cambridge Press, 1988.

[Mel96] J. Melton. An sql snapshot. In *IEEE 12th International Conference on Data Engineering*, New Orleans, Feb 1996.

[Mic94] MicroSoft. *ODBC 2.0 Programmer's Reference and SDK Guide*, 1994.

[Mis93a] P. Missier. Extending a multidatabase language to resolve schema and data conflicts. Master's thesis, Department of Computer Science, University of Houston, September 1993.

[Mis93b] P. Missier. Extensions of MSQL: notes on an implementation. Internal Report, University of Houston, Dept. of Computer Science, March 1993.

[Mon93] S. Monti. Query optimization in multidatabase systems. Master's thesis, University of Houston, December 1993.

[MR95] P. Missier and M. Rusinkiewicz. Extending a multidatabase manipulation language to resolve schema and data conflicts. In *Proceedings of Sixth IFIP International Conference on Database Semantics*, Atlanta, 1995.

[MU83] D. Maier and J. Ullman. Maximal objects and the semantics of universal relation databases. *ACM Transactions on Database Systems*, 8(1):1–14, March 1983.

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specification. In *12th International Conference on Data Engineering*, Feb 1996.

[RC87] M. Rusinkiewicz and B. Czejdo. An approach to query processing in federated database systems. In *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, 1987.

[SANR92] A. Sheth, M. Ansari, L. Ness, and M. Rusinkiewicz. Using flexible transactions to support multidatabase applications. In *US West - NSF - DARPA Workshop on Heterogeneous Databases and Semantic Interoperability*. Boulder, CO, February 1992.

[SK92] A. Sheth and V. Kashyap. So far (schematically) yet so near (semantically). In *IFIP TC2/WG2.6 Conference on Semantics of Interoperable Database Systems*. Elsevier Scientific Publisher B.V., Nov. 1992.

[SM91] M. Siegel and S. Madnick. A metadata approach to resolving semantic conflicts. In *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, September 1991.

[SRL92] L. Suardi, M. Rusinkiewicz, and W. Litwin. Execution of extended multidatabase SQL. Technical Report UH-CS-92-19, Dept. of Computer Science, University of Houston, 1992.

[SRL93] L. Suardi, M. Rusinkiewicz, and W. Litwin. Execution of extended multidatabase SQL. In *Proceedings of the 9th IEEE International Conference on Data Engineering*, Vienna, 1993.

[SS94] E. Sciore and M. Siegel. Using semantic values to facilitate interoperability among heterogeneous information systems. *Transactions on Database Systems*, June 1994.

[SSKS95] L. Shklar, A. Sheth, V. Kashyap, and K. Shah. Infoharness: Use of automatically generated metadata for search and retrieval of heterogeneous information. In *Proceedings of CAiSE-95*, 1995.

[SSKT95a] L. Shklar, A. Sheth, V. Kashyap, and S. Thatte. Infoharness: A system for search and retrieval of heterogeneous information (extended version). In *Proceedings of ACM SIGMOD*, 1995.

[SSKT95b] L. Shklar, A. Sheth, V. Kashyap, and S. Thatte. Infoharness: the system for search and retrieval of heterogeneous information. In *Database Application Semantics, Proc. of the 6th IFIP Working Conference on Data Semantics*, 1995.

[SSR92] E. Sciore, M. Siegel, and A. Rosenthal. Using semantic values to facilitate interoperability among heterogeneous information systems. *Transactions on Database Systems*, 17(12), 1992.

[Sua92] L. Suardi. Execution of extended MSQL. Master's thesis, Department of Computer Science, University of Houston, June 1992.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, march 1992.

[Woo85] William A. Wood. What's in a link. pages 216–241, 1985.