DATA CONNECTIVITY FOR THE COMPOSITE INFORMATION SYSTEM/ TOOL KIT

TOON KING WONG

June 1989 WP # CIS-89-03

Data Connectivity for the Composite Information System/ Tool Kit

Toon King Wong

Bachelor of Science Thesis in Computer Science and Engineering
Massachusetts Institute of Technology
Cambridge, MA 02139

WP# CIS-89-03

ABSTRACT

The Composite Information/ Tool Kit (CIS/TK) is a prototype being developed at the MIT Sloan School of Management for providing connectivity among information systems. At the core of CIS/TK is a distributed database management system called MERGE.

MERGE provides a uniform interface for retrieving and combining data from pre-existing, heterogeneous databases. This is achieved without any additions to the databases or its related programs. Through a global schema, the user is presented with an integrated view of the data. Data is referenced using a common query language called the Global Retrieval Language (GRL). A global query processor executes GRL, and is reponsible for retrieving data from local databases and merging data. MERGE also provides facilities for interfacing with modules whch can handle data reconciliation.

This thesis describes the design and implementation of MERGE. An application for demonstrating MERGE, the Placement Assistant System, is also presented.

KEYWORDS AND PHASES: distributed database management systems, information systems, query processing.

ACKNOWLEDGEMENTS: Thesis advisor: Stuart Madnick; with assistance from the CIS/TK Research Group. Supported in part by the MIT International Financial Services Research Center, Reuters, AT&T, and LCS/IBM.

Table of Contents

Data Connectivity for the Composite Information System/ Tool Kit

1 INTRODUCTION

The *Composite Information System / Tool Kit* (CIS/TK) is a prototype being developed at the MIT Sloan School of Management for providing connectivity among information systems. At the core of CIS/TK is a distributed database management system called MERGE.

MERGE provides a uniform interface for retrieving and combining data from pre-existing, heterogeneous databases as if the data came from a single virtual database. This is achieved without any additions to the databases or its related programs. Through a global schema, the user is presented with an integrated view of the data. Data is referenced using a common query language called the Global Retrieval Language (GRL--pronounced girl). A global query processor executes GRL, and is responsible for retrieving data from local databases and merging data. In addition, MERGE provides facilities for interfacing with modules which can handle data reconciliation.

This thesis describes the design and implementation of MERGE. An application for demonstrating MERGE, the Placement Assistant System, is also presented.

1.1 Background - The CIS/TK Project

With the increasing use of computer-based information systems, the difficulty of combining information and data from various sources is becoming more apparent and has triggered large research efforts toward integrating information systems. We refer to this class of studies and systems as Composite Information Systems.

The CIS/TK project includes a prototype system being developed at MIT using a combination of artificial intelligence, networking and database technology to support connectivity among information systems.

Several issues in realizing connectivity were identified in previous work [MAD 88-1], the technical issues being divided into three levels: physical connectivity, data connectivity and semantic connectivity as represented in Figure 1.1

Semantic Connectivity

Data Connectivity

Physical Connectivity

Figure 1.1 Three Levels of Connectivity

Physical connectivity refers to the ability to physically link and access information systems. However, getting the data is only the first step. In order to be useful, the data has to be merged and formatted into a manageable form. This ability is referred to as data connectivity. Data from multiple and different sources often have data conflicts such as contradiction, ambiguity and incompleteness. Semantic connectivity refers to the ability to reconcile these inconsistencies using knowledge captured from the user about the assumptions underlying the data.

The goal of CIS/TK is to develop tools and techniques to support the entire spectrum of connectivity, with a focus on semantic connectivity. The CIS/TK approach [MAD 88-2] explicitly allows for the coexistence and usage of a variety of information systems while preserving their local autonomy. These information systems are typically independently developed, hard to modify, and contain data that is dynamically changing.

1.2 Data Connectivity for CIS/TK

Recent developments in CIS/TK have aimed at developing an integrated system for the MIT Sloan School Student Placement Office, allowing integrated access to several databases as Figure 1.2 shows [WAN 88-1]. This thesis focuses on providing data connectivity among the databases; allowing users to access and combine data from the various dissimilar databases as if the data came from a single virtual database. In addition, although this thesis does not explicitly address issues involved in providing application development mechanisms like expert systems, and knowledge base management systems for resolving semantic conflicts, one of the major objectives is to provide an environment and foundation with which research in semantic connectivity can be investigated.

1.3 Goals of Thesis

In order to provide data connectivity for CIS/TK, MERGE must achieve the following goals:

(a) Provide a common data model for viewing the underlying data,
(b) Provide facilities for processing a common query language, and
(c) Serve as a foundation for semantic connectivity research.

In addition, an application called the Placement Assistant System was developed to demonstrate the feasibility of MERGE.

1.4 Overview of Thesis

The focus of this thesis is in the design of a distributed database management system for CIS/TK.

In Chapter 2, we present some related work in distributed database management systems, and present the approach we adopted in developing MERGE.

In Chapter 3, we present an overview of the MERGE architecture and also some of the major design considerations in developing the system.

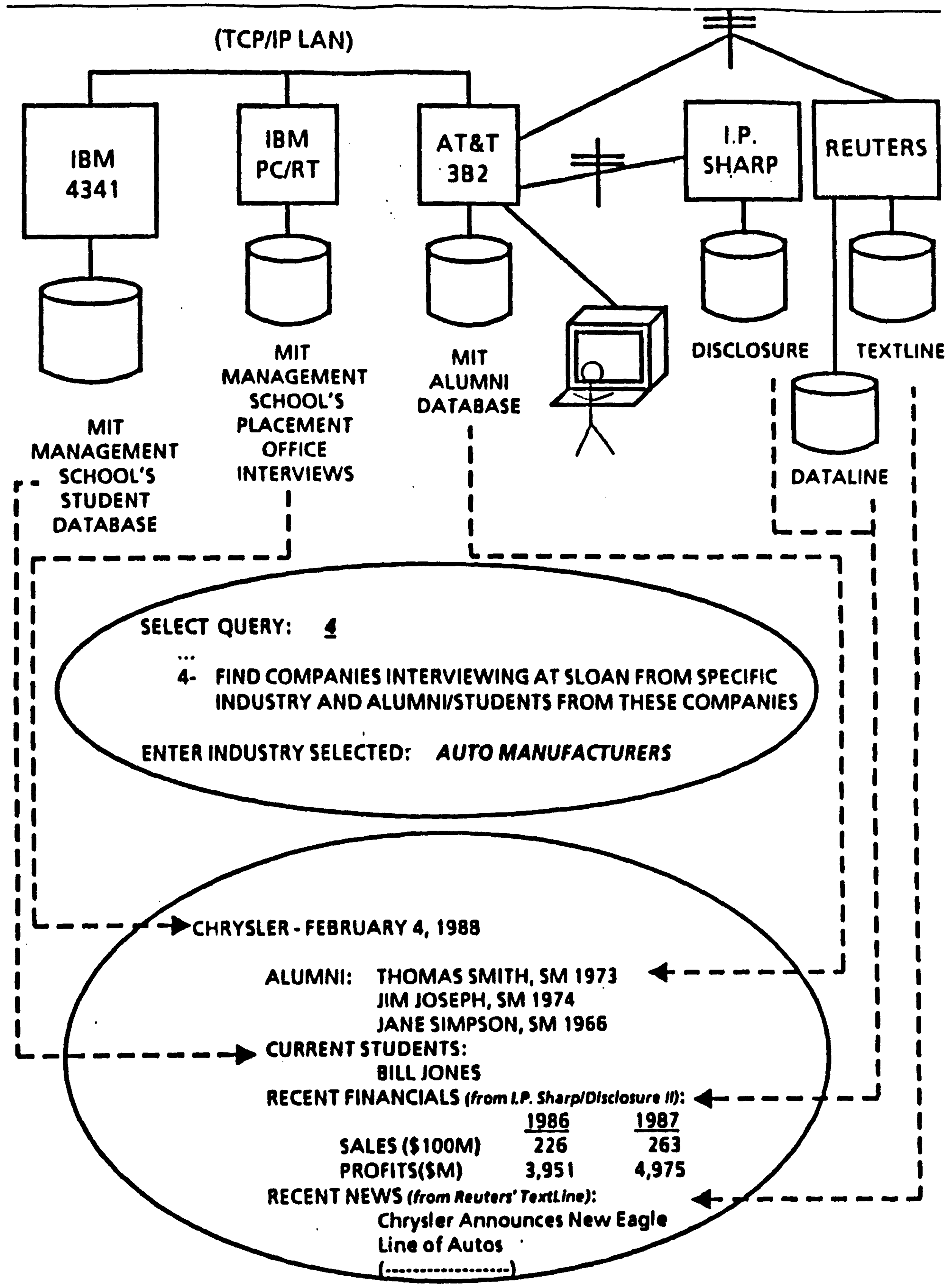
In Chapter 4, we present an overview of the Local Query Processor, which provides a uniform method of retrieving data from dissimilar databases.

In Chapter 5, we present the MERGE Data Model, which presents a single, integrated view of the underlying data.

In Chapter 6, we present the Global Query Processor, a facility for processing the common query language GRL.

In Chapter 7, we describe an application, called the Placement Assistant System, to demonstrate the feasibility of MERGE.

Finally, in Chapter 8, we present our conclusions about the design of MERGE and suggests some future work.



(TCP/IP LAN)
IBM 4341
IBM PC/RT
AT&T 3B2
I.P. SHARP
REUTERS
MIT MANAGEMENT SCHOOL'S STUDENT DATABASE
MIT MANAGEMENT SCHOOL'S PLACEMENT OFFICE INTERVIEWS
MIT ALUMNI DATABASE
DISCLOSURE
TEXTLINE
DATALINE
SELECT QUERY: 4
...
4- FIND COMPANIES INTERVIEWING AT SLOAN FROM SPECIFIC INDUSTRY AND ALUMNI/STUDENTS FROM THESE COMPANIES
ENTER INDUSTRY SELECTED: AUTO MANUFACTURERS
CHRYSLER - FEBRUARY 4, 1988
ALUMNI: THOMAS SMITH, SM 1973
JIM JOSEPH, SM 1974
JANE SIMPSON, SM 1966
CURRENT STUDENTS:
BILL JONES
RECENT FINANCIALS (from I.P. Sharp/Disclosure II):
1986 1987
SALES ($100M) 226 263
PROFITS($M) 3,951 4,975
RECENT NEWS (from Reuters' TextLine):
Chrysler Announces New Eagle Line of Autos
(-----------------)

Figure 1.2 Connectivity for the MIT Management School's Placement Office

2 RELATED RESEARCH

Systems that provide data connectivity between databases are generally categorized as distributed database management systems. A large part of this thesis draws upon work done in this area, in particular the Multibase system [ROS 82].

2.1 Approaches to Integration

Depending on the application and the constraints, there are several approaches to the development of distributed database systems. However, most of these systems can be broadly distinguished on two aspects: heterogenuity and control [DEE 82].

Heterogeneous Vs Homogeneous
Homogeneous systems support one data model and one manipulation language. Unfortunately, these systems cannot meet the objectives of most organizations who use many types of computers with different data models and multiple data manipulation languages.

To meet such objectives, it is necessary to use a heterogeneous system. Heterogeneous distributed databases access and manipulate information maintained in existing, distributed, heterogeneous DBMSs through a single uniform interface. This is accomplished without changing existing database systems and without disturbing local operations.

Centralized vs Decentralized
In a centralized system, all global processing is controlled by a central computer. The disadvantage of this approach is that it creates a bottleneck and reduces the stability of the system, since the failure of the central computer disables the distributed database system.

In a decentralized system, each node keeps a copy of the distributed database system, each supervising the global transactions submitted from it. The system is more stable, since the breakdown of a single node does not disable the whole distributed system. However, the exercise of controls and the preservation of consistency is more difficult.

For MERGE, we opted for a centralized, heterogeneous system. The main reason is because Merge is designed to support different databases which are not wholly under the control of any one organization. In MERGE, all components of the distributed DBMS reside on the central computer. No additions or changes to the local databases or their host systems are required.

2.2 Issues in Heterogeneous Distributed Systems

The major issues faced in developing Distributed Heterogeneous Database Management Systems (DHDBMS) include [BHA 87]:

(a) Developing a Common Data Model,
(b) Providing facilities for Query Processing,
(c) Incorporating Distributed Transaction Management Routines, and
(d) Developing Authorization and Control Data Security Procedures.

Since MERGE is presently designed to perform retrieval-only operations, the problems of transaction control, and data security are not major factors. Instead, this thesis only focuses on the issues of developing a common data model and providing facilities for query processing.

(a) Common Data Model
The goal of a common data model is to capture the entire meaning of the underlying data. In order to achieve this, it has to resolve data conflicts resulting from the integration of different systems and dissimilar data models.

Data conflicts can be distinguished into two types: structural and semantic. Structural conflicts include differences in data models and differences in implementation of the local databases. Semantic conflicts include differences in naming, data representation, and data scaling. Most work in DHDBMS address the resolution of structural conflicts. However, very few aim at resolving semantic conflicts.

As with most DHDBMS, MERGE adopts a three schema approach in data integration: a conceptual schema, an internal schema, and an external schema. A conceptual schema defines all the data in the environment, which is mapped to many underlying file and DBMS structures; referred to as the internal schema. The conceptual schema is also mapped to many user views; which is referred to as the external schema. The use of multiple schemas and the mappings between them serves as the mechanism for providing transparency across dissimilar systems and architectures.

(b) Query Processing
Query processing and optimization are complicated by the following factors:

(a) Multiple sources for data,
(b) Different local processing capabilities at the local database management systems,
(c) Different communication costs, and
(d) Variable speeds of communication links.

Most DHDBMS have optimizations for more efficient query processing. These strategies include various join strategies, submitting subqueries to DBMS in parallel, parcelling out as much computation to individual DBMS, and selecting access paths which provide optimal returns in communication costs and speed.

In MERGE, the query processing facilities provide a uniform interface for retrieving data from various DBMS. Presently, optimizations in the query processor are relatively simple and are focused only on retrieving data in a reasonable space of time. Optimizations include the automatic creation and selection of access paths using a changeable set of rules, and a join strategy aimed at narrowing the search space.

2.3 MERGE as a Foundation for Semantic Connectivity

As described in the previous sections, the issues involved in designing MERGE are similar to the problems found in developing distributed database management systems. However, what distinguishes MERGE from these systems is the fact that it is intended to serve as the foundation for further work in semantic connectivity.

This major objective has influenced us in the various stages in the design of MERGE and CIS/TK. MERGE must be extensible and provide interfaces to tools that can resolve semantic conflicts in the data. Several such tools proposed include inter-database instance identification [HOR 88], translation facilities for resolving scale conflicts [MCC 88] and concept inferencing [WAN 88-2]. In contrast, most work in distributed database management systems ends at the data connectivity level [NEU 82] [LIN87].

3 OVERVIEW OF MERGE

This chapter provides an overall view of the MERGE architecture and also discusses some of the major design considerations in the development of MERGE. The intent of MERGE is to lay the foundation for further research in connectivity, in particular, research in semantic connectivity. Through a global query, MERGE will provide the ability to retrieve data from disparate databases as if the data came from a single database, and thus allow researchers to concentrate on the more challenging issues found in data reconciliation. As the CIS/TK project is one that is continuously evolving, the ability to extend the components within CIS/TK without major modifications is a critical design goal. This key goal strongly influenced us at all stages in the development of MERGE.

In this chapter, we first present the overall architecture of CIS/TK and its relation to MERGE. Then we address some of the problems faced in representing a single, integrated view of the data. In Section 3.3, we describe some of the different types of problems in data reconciliation. Then in Section 3.4, we describe the implementation environment of CIS/TK and MERGE. Lastly, we describe some of the major problems found in the previous prototype of CIS/TK, and how MERGE intends to solve these problems.

3.1 Data Connectivity for CIS/TK

A key component within the CIS/TK system is the query processing facility which controls the execution of queries. The query processing architecture [HOR 88-2] is divided into three levels, each level providing some aspect of connectivity. Figure 3.1 shows the query processing architecture of CIS/TK and how MERGE is related to the various components within CIS/TK. In the following sections, we briefly describe the various levels of the query processing architecture.

3.1.1 The Local Query Processor

The lowest level of the query processing architecture is the Local Query Processors (LQP), which provide physical connectivity to various local DBMS. Each LQP handles communications to a single local database and with the computer on which the database resides. The LQP provides a uniform interface for the GQP to access dissimilar databases, handling the particularities of each local DBMS and its host system.

3.1.2 The Global Query Processor

The middle level is the Global Query Processor (GQP), which provides data connectivity through a global query language and a common data model. The GQP is responsible for parsing a global query and routing the subqueries to the appropriate LQPs' for data retrieval. After the LQPs return the data, it is combined and returned to the AQP level.

A major component of MERGE is the GQP. In our version of MERGE, the GQP is further divided into a parser module and a router module.

3.1.3 The Application Query Processor

The top level is the Application Query Processor (AQP), which provides for semantic connectivity by using the domain of the application to resolve conflicts found in the data. The AQP is responsible for mapping the application query into an equivalent global query for retrieving data. Presently, the AQP level is still a subject of initial research, so we will not describe it further.

Of these three levels of query processing, MERGE implements the middle level which includes a global query processor and its associated data model. These components will be further described in Chapters 5 and 6.

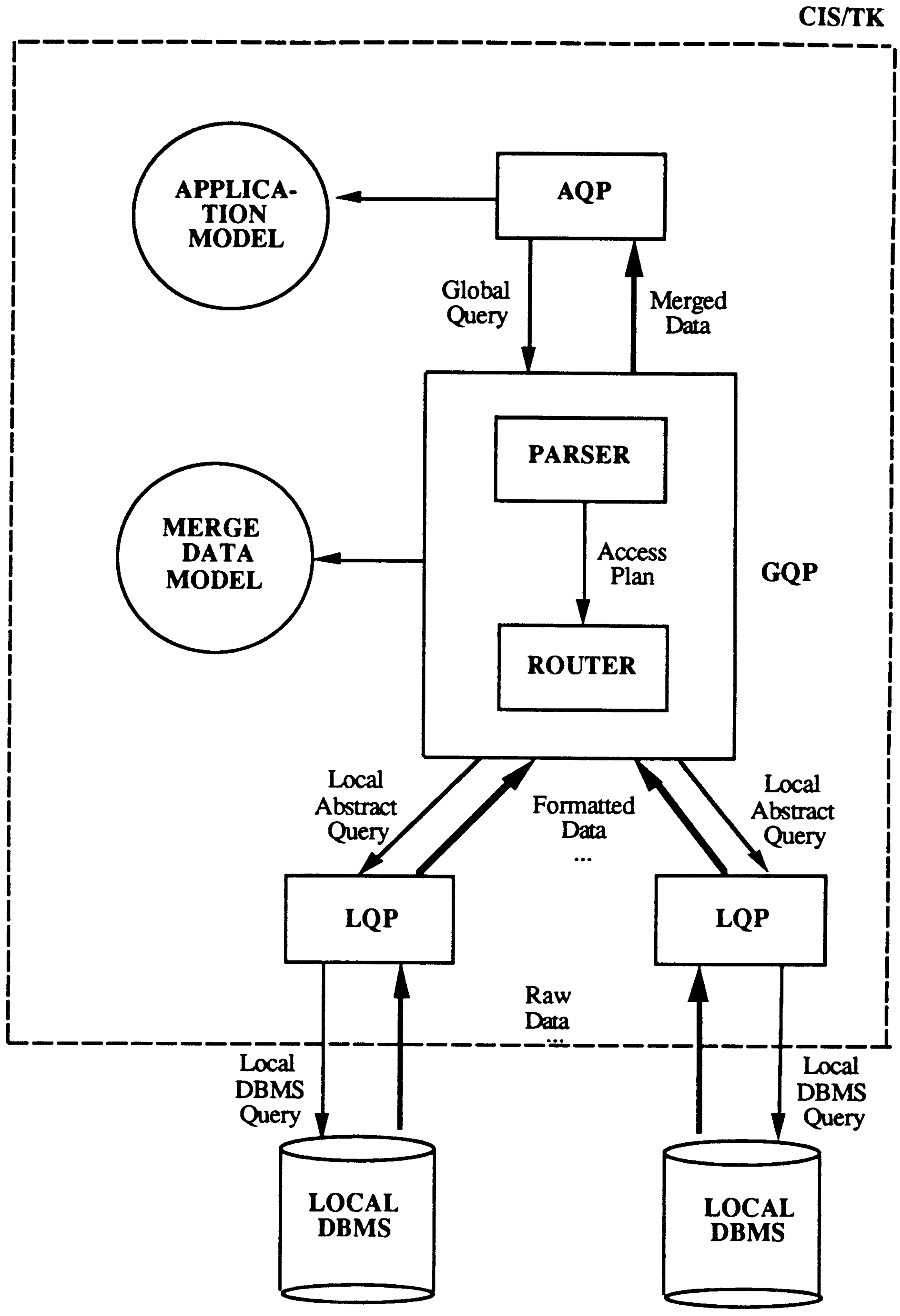


Figure 3.1 The CIS/TK Architecture and MERGE

3.2 Structure and Data Representation

Representating a single, integrated view of all the data in a distributed database system is especially challenging because of the dissimilar structures adopted by each local DBMS. As with most other DDBMS, CIS/TK adopts a three-schema architecture for representing data, as shown in Figure 3.2. The internal schemas are created by the local DBMS and are assumed to be pre-existing. Merge implements the middle schema, that is the data model, which represents a single, integrated view of the data. The application model is implemented at the AQP level, and represents a subset of the data necessary for a particular application. The application model may also represent data not explicitly available in the underlying databases, but which may be derived or deduced from that data.

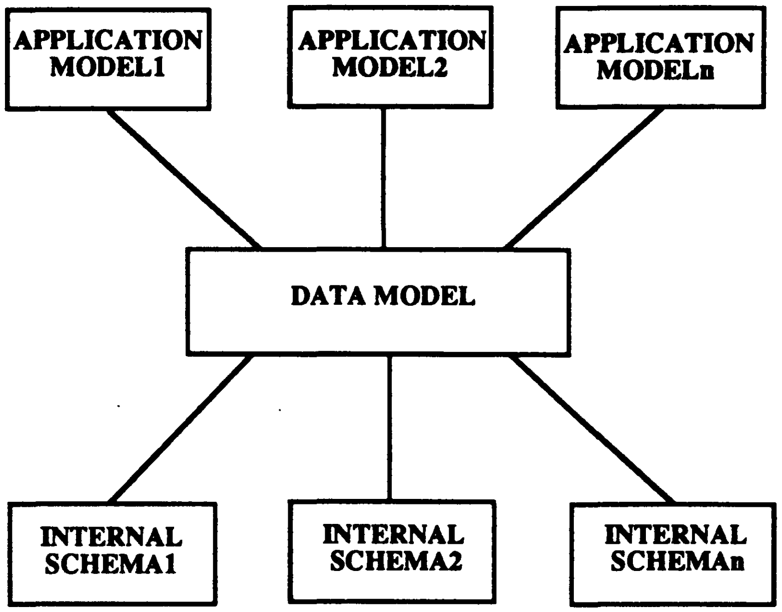


Figure 3.2 The CIS/TK Three-schema Architecture

In MERGE, the structural properties of the data are distinguished from the semantic properties of the data. In the data model, the structural properties of data including attribute names and relationship between tables are represented by a global schema. On the other hand, the semantic properties of data including synonyms and translations between different data representations are a data catalog.

The main reason for separating the structural properties from the semantic properties is because in the near future, we would like to extend and enhance the semantic representation capabilities of our data model to incorporate schemes to represent conflicts in inter-database identification and better schemes for representing synonyms and translations. An integrated representation scheme would make these extensions harder to achieve.

3.3 Data Reconciliation

Combining data from disparate sources is difficult because the data are often found in different formats, and different representations and is usually contradictory and incomplete. In order to combine data, MERGE provides certain necessary data reconciliations.

3.3.1 Types of Data Conflicts

Conflicts in data can be distinguished as two types: syntax conflicts and semantic conflicts. Syntax conflicts are obvious conflicts like differences in naming, formats, and scale representations. For example, in a recruitment database shown in Figure 3.3, the same company may be called several different names, like "Ford Motors" and "The Ford Motor Company". We refer to these similar names as synonyms. Although they represent the same concept, they are spelled differently in the data. In contrast, semantic conflicts are more subtle.

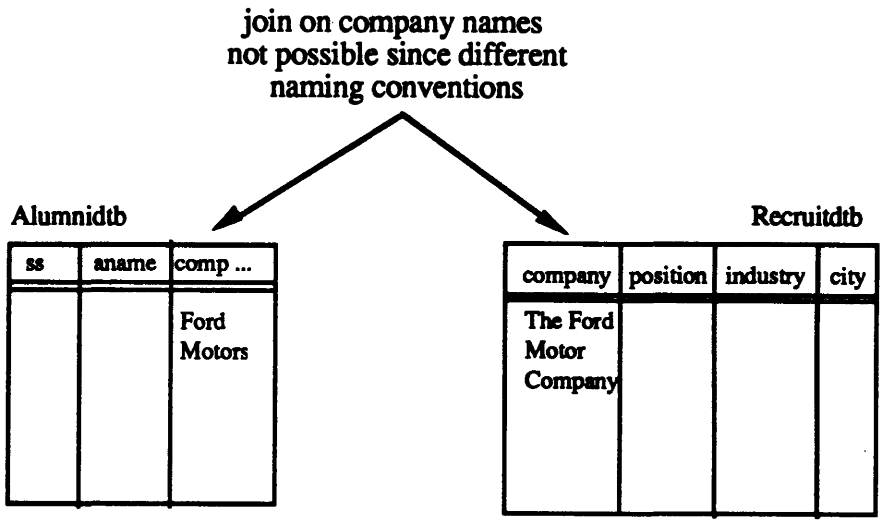


Figure 3.3 An Example of Difference in Naming

Semantic conflicts include differences like contradiction, incompleteness and ambiguity which arise because the local databases were independently developed, and often carry quite different assumptions about the data. A good example is financial databases. Financial data like a company's revenue or net income are often calculated based on the practices of the country where the company is located or incorporated. Thus, to match the performance of two companies based on the revenue data may be misleading because of these different assumptions for calculating revenue.

Unfortunatlely, it is not within the scope of this thesis to detail the different data conflicts found in the real world, and the reader is referred to the works of [WAN 88-3], [PAG 89], which present interesting examples found in the hotel and financial industry. Nevertheless, MERGE has been designed as a basis for future more detailed research on semantic conflicts.

3.3.2 Resolving Conflicts in MERGE

Resolving syntax conflicts, although tedious, is not as difficult as resolving semantic conflicts. Resolving semantic conflicts require an in-depth knowledge of the domain of the application and requires special tools and techniques for the representation of the domain knowledge and for applying this knowledge to data reconciliation. These issues are addressed at the AQP level with tools like the application model and concept inferencing. At the GQP level, only syntactic conflicts are addressed like differences in naming, formats and scale representations. In this version, we will only handle differences in naming.

MERGE provides a data catalog system to represent synonyms, and interfaces to modules which make use of the catalog for data reconciliation. The data catalog is further described in Chapter 5, and the interfaces are described in Chapter 6.

By considering data reconciliation in the development of MERGE, it is possible to design an architecture that can accomodate future extensions of facilities for data reconciliation.

3.4 Implementation Environment

CIS/TK is being developed on a UNIX platform to take advantage of its portability across disparate hardware, its multi-tasking environment, and its communication capabilities to enable access to multiple remote databases in concert. The kernel of CIS/TK is being developed using KOREL [LEV 87], an object-oriented programming language developed in the Common Lisp environment.

Using KOREL, we are able to benefit from the features of the object-oriented paradigm [WEG 86] -- modularity, consistent interfaces and conceptual clarity. Because MERGE is designed to work within CIS/TK, it is also developed using the object-oriented paradigm. However, for efficiency reasons, only the major interfaces in MERGE use KOREL, the other components are developed in LISP, which unlike KOREL, does not incur the extra cost of message-passing.

3.5 Improvement to Prototype

A preliminary prototype of CIS/TK was developed in previous work [WON 88]. Insights gained from the prototype and from a financial application [PAG 89] were helpful in the design of MERGE.

At the global query processing level of the earlier prototype, there was general dissatisfaction with the query language in its readability. In addition, selection of the numerous databases to satisfy a global query had to be manually performed. This proved to be frustrating to users who were unfamiliar with the underlying database configuration.

In MERGE, an improved SQL-like query language was developed. Since SQL [DAT 87] is fast emerging as the de-facto standard for database query languages, users are likely to be more receptive to the new query language. Also, an innovative database selection mechanism that automatically selects the databases for a global query was developed. Another feature of the selection mechanism is that it relies on a set of changeable parameters for determining the criteria for database selection, in contrast to most other optimized mechanisms [ROS 82], where the criterias are imbedded within the mechanism itself, making it difficult to change. These improvements are described further in Chapter 6.

At the conceptual schema level, several inconsistencies in the data model detracted users from a clear understanding of the model. Some of these inconsistencies included differences in the representation of relations and fragments. In the design of MERGE, these issues were addressed and are discussed in Chapter 5.

This chapter provided an overview of MERGE and the major design considerations in developing MERGE. In the next chapter, before presenting the main components of MERGE, we provide an overview of how the GQP can retrieve data through a Local Query Processor. Although the LQPs are not a focus of this thesis, they provide the ability for MERGE to retrieve data from dissimilar databases on various host machines through a common interface.

4 LOCAL QUERY PROCESSING

To access a database, the Global Query Processor relies on the Local Query Processor (LQP) to perform the actual physical connection, and retrieval of data from the database host machine. Each database that is to be accessed by CIS/TK must have an LQP. These LQPs reside on the CIS/TK host machine and not on the database host machines. In this chapter, we provide a brief overview of how data retrievals can be accomplished through the LQP. For a detailed description of how the various LQPs work, please refer to [CHA 88], [GAN 89], [GER 89].

4.1 Retrieving Data Through the LQP

The LQP provides a uniform method of connecting and retrieving data from various databases using a query language called the Abstract Query Language. The basic structure of an AQL query is:

(send-message *lqp* :get-data (*table* (*att1 att2 ... attn*)) *conditions*)

Figure 4.1 shows an LQP processing an AQL query to a SQL-based DBMS. The AQL query is translated by the LQP into an SQL query and executed at the local DBMS. The raw data from the DBMS is typically returned as a file, which the LQP reformats into a data list with the following format:

((att1 att2 ... attn)
 ("val1" "val2" ... "val3") ... ("val1" "val2" "val3"))

where the first list contains the attribute names, and the rest of the list contains the values corresponding to those attributes.

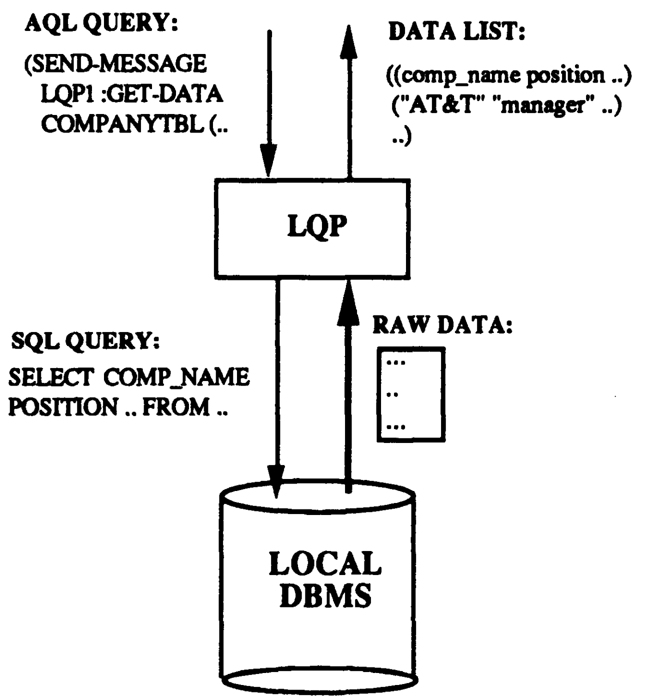


Figure 4.1 Retrieving Data Through The LQP

Presently, the databases supported by LQPs include several SQL databases on AT&T 3B2 UNIX machines, and an IBM/RT XENIX machine. Also planned in the near future is the completion of two LQPs to support retrievals from commercial financial databases, which are menu-based systems rather than SQL-based systems.

5 THE MERGE DATA MODEL

The MERGE Data Model (MDM) serves as the conceptual basis for viewing the distributed database system -- it provides a single, integrated view of the underlying data. The data model is implemented through three components: a global schema, a data catalog, and a query language called the Global Retrieval Language (GRL). These components are used by the Global Query Processor for processing a global query. For the reasons mentioned in Chapter 2, data representation in the MDM distinguishes between the structural properties and the semantic properties; the structural properties are represented by a global schema and the semantic properties by a data catalog.

In this chapter, we discuss the problems in representing a single, integrated view of data in a multi-database environment, and present how the global schema, the data catalog and the GRL address these issues.

5.1 The Global Schema

The objective of a global schema is to represent the structures and relationships in the underlying data. The global schema uses an extended version of the Entity-Relationship (E-R) model [CHE 76] to describe these structures, chosen because it is widely accepted in database design and simple to understand.

Figure 5.1(a) shows a simple global schema created to represent data available from two sources: a recruiting company database and an alumni database. The underlying databases are shown in Figure 5.1(b). The global schema has two entities: the *alumni* entity and the *company* entity. The *alumni* entity represents all the data about alumni, and the *company* entity representes all the data about companies that are recruiting. The entities are related on the relationship *works_for*, which represents the fact that the alumni information can be joined to the company information using the company names found in both the recruit and alumni databases. We will describe this global schema further when we address the problems in schema integration.

To provide a single integrated view of the data, the dissimilar schemas of the local databases have to be integrated. In Section 5.1.1, we discuss the major issues that are faced in schema integration, and present how the global schema addresses these problems. To implement a global schema, we found it necessary to develop a schema definition language to describe the global schema. This is outlined in Section 5.1.2.

5.1.1 Issues in Schema Integration

Some of the major issues in schema integration include resolving problems in:

(a) attribute naming,
(b) attribute organization,
(c) fragmentation,
(d) multiple relations, and
(e) complex relations.

(a) Attribute Naming

In a multi-database environment, similar attributes are often found with different names. In order to present a unified view of the data, similar attributes with different names have to be resolved.

In the global schema, this is handled by assigning a global attribute name to local attributes that represent the same thing. For example, the *company* entity in our example has a global attribute called *name*. This actually represents two local attributes found in the tables *east_companytb* and *west_companytb*, called *company* and *comp_name* respectively. As a convention, we will address global attributes and local attribute in the following manner:

(entity attribute)   - unique identifier for global attribute
(lqp table attribute) - unique identifier for a local attribute

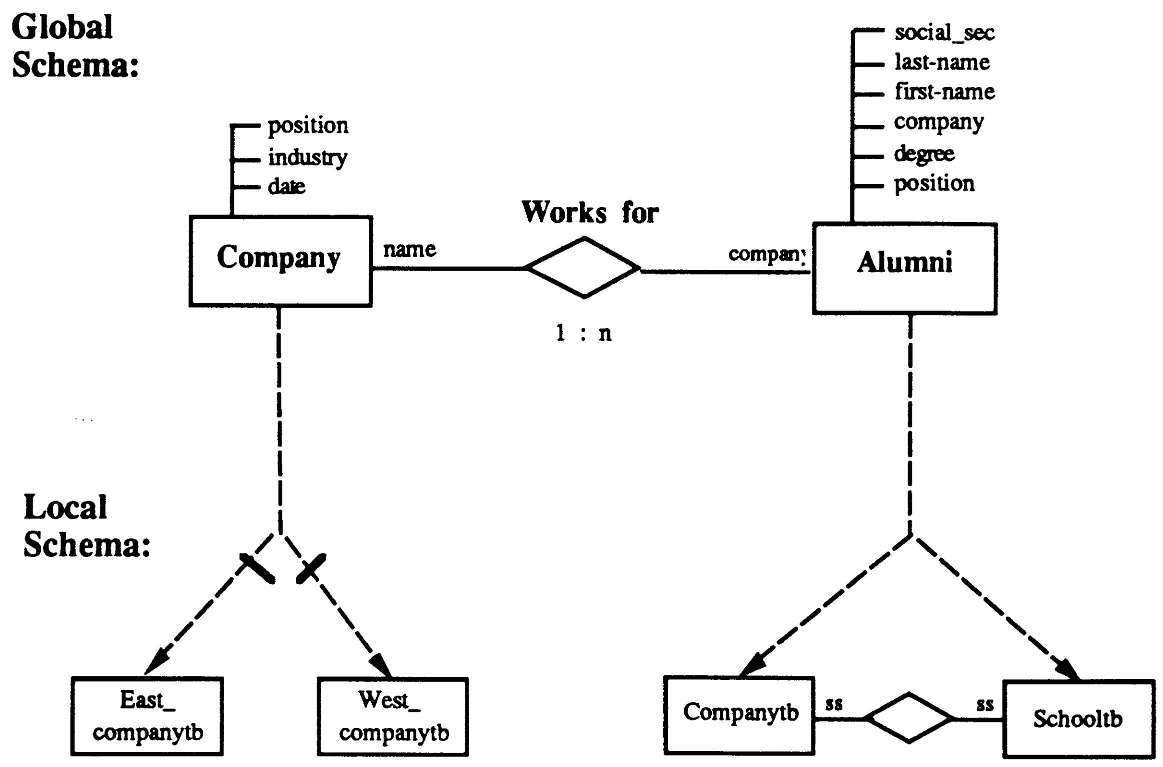


Figure 5.1(a) Simple Placement Global Schema

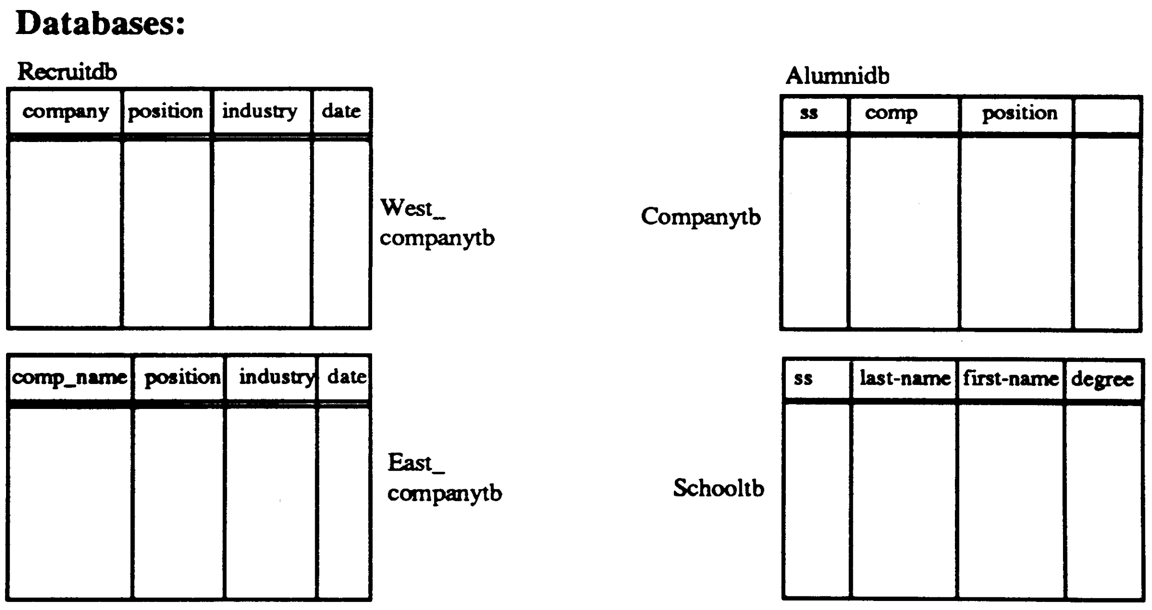


Figure 5.1(b) Underlying Databases

Note that for the unique identifier for a local attribute, the LQP name is used instead of the database name. This is because since each LQP is responsible for accessing one database, it is equivalent to the database name for identification purposes. In addition, within MERGE, accessing data is through the LQPs, so this provides a means of invoking the appropriate LQP for a local attribute. In our examples, we will assume that the LQPs have the same names as the databases.

(b) Attribute Organization

Attribute organization refers to the grouping of attributes in entities. Attribute organization is mostly subjective; attributes are grouped into an entity because they represent a common concept. However, there is one constraint in the global schema that has to be adhered to. For example, in the *simple-placement* global schema, the entity *alumni* has attributes like *major*, *degree* and *position*; which are attributes commonly associated with an alumni. One attribute that is not so clearly defined is *(alumni company)*. This attribute could also be placed in the *company* entity, since it is directly related to information about companies. In fact, within the *company* entity, there is an equivalent attribute called *(company name)*. However, in our global schema, a decision was made not to merge these two attributes. There are two main reasons for this choice.

In the global schema, in order to express a relationship between two entities, they must have at least one similar attribute. In the relationship between the *alumni* and *company* entities this relationship is expressed as:

```
(=  (alumni company)  (company name) )
```

Another more important reason is that it provides a better view of the underlying data structures. The fact that company name is represented in both entities implies that this attribute can be found in at least two databases; the alumni and the recruiting database. This affords us a conceptually clearer view of the underlying data.

(c) Fragmentation

There are basically two types of fragmentation found in databases: horizontal fragmentation and vertical fragmentation. Vertical fragmentation is the separation of data by domain, for example in the recruiting database, data about recruiting companies is divided into companies that are from the West Coast, and companies that are from the East Coast. On the other hand, horizontal fragmentation is the separation of data by attribute values, for example in the alumni database, the attributes for an alumni are divided between two tables: school information like *degree* is found in *schooltb*, and the alumni's company information is found in *companytb*.

In reality, resolving fragmentation is difficult because data is typically overlapped with both horizontal and vertical fragments even within a single table. Most integration schemes do not addresss the issue of overlapping fragments. In the global schema, we will address only non-overlapping fragmentation, leaving the issue of overlapping fragmentation as future work.

In the global schema, the purpose is to integrate these fragments. We integrate fragments by expressing the relationships that exist amongst the fragments. Vertical fragments are expressed as a merge, and horizontal fragments are expressed as a concatenation. For example, to integrate the fragments in the alumni database into a single entity called *alumni*, we have to express the following relationship between the tables found in the alumni database:

```
(merge  (alumnidb  schooltb)  (alumnidb  companytb)
        on  ( =  (alumnidb  schooltb  ss)  (alumnidb  companytb  ss)))
```

which means that in order to get data that spans across the tables (fragments) *schooltb* and *companytb*, we need to merge those two tables on the social security local attribute, since the social security is the common attribute between those two tables. In the above relation, the table *schooltb* is represented as *(alumnidb schooltb)* so that we can uniquely identify the table that we are refering to.

To represent a vertical fragment, we use the idea of a concatenate. For example, to integrate the tables in the recruiting database into a single entity called *company*, we express the following relationship:

```
(concatenate  (recruitdb   west_companytb)
                (recruitdb   east_companytb))
```

which means that in order to get all the companies represented by the *company* entity, we have to concatenate the data found in *west_companytb* to the data found in *east_companytb*.

(d) Multiple Relationships

There is usually more than one way to draw relations between data. For example, in the company and alumni entities, the works_for relationship expresses a join between the company names. However, there is yet another possible join between those two entities; between *(company position)* and *(alumni position)*. A good representation scheme must be flexible enough to allow for the expression of multiple relationships.

In the global schema, multiple relationships between entities can be expressed in a rather straightforward manner. To express the join:

```
(=  (company   position)  (alumni   position)
```

we can draw another relation, same_position between the entities as shown in Figure 5.2.

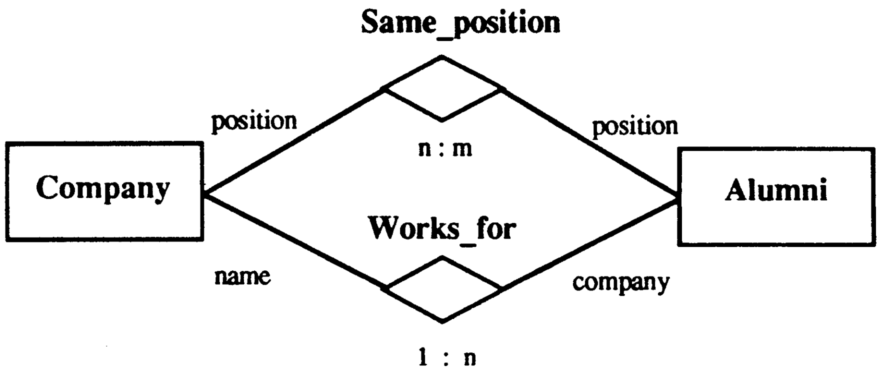


Figure 5.2 Expressing Multiple Relations in the Global Schema

(e) Complex Relationships

In some cases, the relationship between two tables is not simply a join between 2 attributes, but instead involves several attributes. For example, consider a database containing the phone bills and the addresses of telephone owners as shown in Figure 5.3. Each table is uniguely identified by the telephone number, which is separated into two fields: *area-code* and *7digits*. In order to join between the two tables to get all information about a phone owner, the tables have to be joined on both the *area-code* and *7digits* attributes.

In previous prototypes of the global schema, complex relationships were not supported. However, in the financial application built by [PAG 89], we found that such complex relationships commonly exist. In this version, we have designed the global schema to support complex relationships between entities by using the following predicate syntax:

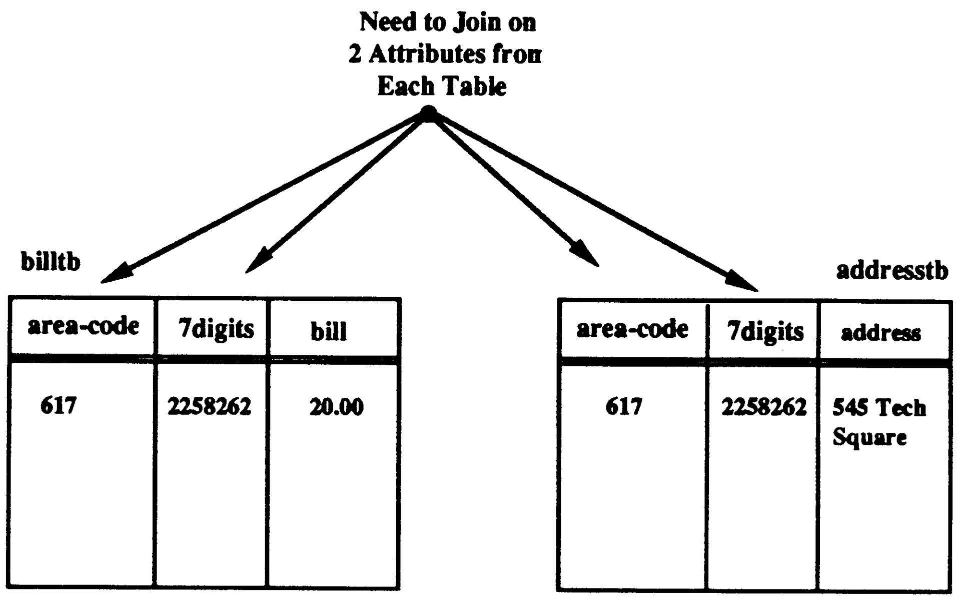


Figure 5.3 Complex Relation Between Tables

(and ***cond1 cond2*****)**

For example, to represent the relationship between the two tables in the phone database, the following expression is used:

```
(and (= (billtb  area-code) (addresstb  area-code))
       (= (billtb  7digits) (addresstb  7digits))
```

Our solution for handling complex relationships touches only the surface of the problems found in representing relationships. Other possible predicates could include the operator "or" and condition predicates like ">" and "<". We leave the idea of creating a general set of relation operators that can accomodate different types of relations as future work.

5.1.2 An Overview of the Schema Definition Language

In the previous section, we have presented the global schema and how it addresses some of the major issues in schema integration. In this chapter, we describe the language used to implement a global schema, called the schema definition language. The E-R model has traditionally been used for conceptual schema design. Presently, no standard language for implementing an E-R model schema exists. In MERGE, a schema definition language has been developed for implementing the E-R model.

Using an object-oriented paradigm, entities and relationships may be viewed as objects. The schema definition language allows for the creation of these entity and relationship objects. The schema definition of the *simple-placement* global schema is shown in Figure 5.4 The following sections give an overview of how to create entity and relation objects.

Creating a Global Schema

To create a global schema, the *create-schema* statement must be placed at the beginning of the file before creating any entity or relation objects. The format used is:

```
(create-schema   name )
```

Creating Entities
To create an entity, the *create-entity* statement is used. This statement has the following syntax:

```
(create-entity  name
                :attributes   ((gatt1   loc1   ... locn)  ;; gattn - global attribute name
                               ...                         ;; locn - (lqp tb col)
                                (gattn  loc1  ... locn))
                :table-relations  ((merge  source1   source2    ;; sourcen - (lqp tb)
                                          on  cond)
                                   (concatenate source1  source2)
                                    ... ))
```

The statement has two slots. The *:attributes* slot is used to assign global names for similar attributes found in the local databases. The *:table-relations* slot is used to express relationships between various fragments (tables) represented by the entity.

Creating Relations
To create a relation, the *create-relation* statement is used. Before creating relations between entities, the entities must be created first because the *create-relation* statement checks for the existence of these entities before creating a relation object. The basic syntax of the create-relation statement is:

```
(create-relation  name
                  :entity-from   entity
                  :entity-to   entity
                  :join   (=  (entity   att ) (entity   att))
```

The *:entity-from* and *:entity-to* slots specify which entities are being joined. The *:join* slot specifies the attributes that are being joined on between the two entities.

This section has given a brief overview of the schema definition language. Please refer to Appendix B.2 for a specification of the schema definition language.

```
;;;; This file implements the simple-placement global schema


;;; place at beginning
;;; creates schema
(create-schema simple-placement)

;;; create company entity
(create-entity company
            :attributes  ((name  (recruitdb west_coastb company)
                                 (recruitdb east_coasttb  comp_name))
                          (position  (recruitdb west_coasttb position)
                                     (recruitdb east_coasttb position))
                          (industry  (recruitdb west_coasttb industry)
                                     (recruitdb east_coasttb industry))
                          (date  (recruitdb west_coasttb date)
                                 (recruitdb east_coasttb date)))
            :table-relations  ((concatenate (recruitdb west_coasttb)
                                            (recruitdb east_coasttb))))

;;; create alumni entity
(create-entity alumni
            :attributes   ((social_sec (alumnidb companytb ss)
                                       (alumnidb schooltb ss))
                           (last-name  (alumnidb schooltb last-name))
                           (first-name (alumnidb schooltb first-name))
                           (company  (alumnidb companytb comp))
                           (degree  (alumnidb schooltb degree))
                           (position (alumnidb companytb position)))
            :table-relations  ((merge (alumnidb companytb)
                                      (alumnitb schooltb)
                                    on (= (alumnidb companytb ss)
                                          (alumnidb schooltb ss))))

;;; create works_for relation
(create-relation works_for
            :entity-from alumni
            :entity-to company
            :join (=  (alumni company) (company name)))
```

Figure 5.4 Schema Definition for *simple-placement* Global Schema

5.2 The Data Catalog

The previous section presented how MERGE represents the structural properties found in the underlying data. In this section, we introduce the data catalog, used to express the semantic properties of the data. In this impementation, only one kind of semantic property is represented: synonyms.

5.2.1 Representing Synonyms

The Idea

Synonyms are represented using a catalog that keeps a list of all synonyms for an attribute. For example, the basic structure of a synonym catalog for the *(company name)* attribute is shown in Figure 5.5. The first column contains the main attribute value, which serves as the unique identifier for the synonyms in each row. For example, a main attribute is "IBM", which is a unique identifier for "I.B.M." and "International Business Machines".

main attribute	**syn1**	**syn2**	**syn3 ...**
IBM	I.B.M	...	International Business Machines
DEC	DEC Inc.	...	Digital Equipment Corporation

Figure 5.5 A Synonym Catalog for Company Names

Problems with One-level Scheme

However, there is a problem with this basic scheme. By representing synonyms at the global attribute level, we assume that the synonyms are shared across all the local attributes represented by that global attribute. For example, the global attribute *(company name)* represents two actual local attributes: *(recruitdb west_companytb company)* and *(recruitdb east_companytb comp_name)*. By using the above scheme for representing synonyms, both these local attributes are assumed to have, for example, "IBM" as the main attribute for "International Business Machines" and "I.B.M." In some cases, this assumption is not correct.

Suppose "IBM" represents a different company in each table. Refering to Figure 5.6, "IBM" in *east_companytb* represents "Itsy-Bitsy Machines" and "IBM" in the *west_companytb* represents "International Business Machines." The one-level scheme does not allow us to represent this difference of names at the local database level. In order to represent these differences, we have developed a two-level scheme for representing synonyms.

A Two-Level Scheme

As shown in Figure 5.6, the synonym catalog consists of a single global synonym table and several local synonym tables. The global synonym table contains local attributes that have synonyms, and for each local attribute also contains a pointer to the local synonym table. For example, in the global synonym table **global_syntb**, the attribute *(recruitdb west_companytb name)* has a pointer to the local synonym table **west_syntb**. Each local synonym table contains the actual synonyms for each local attrinute. For

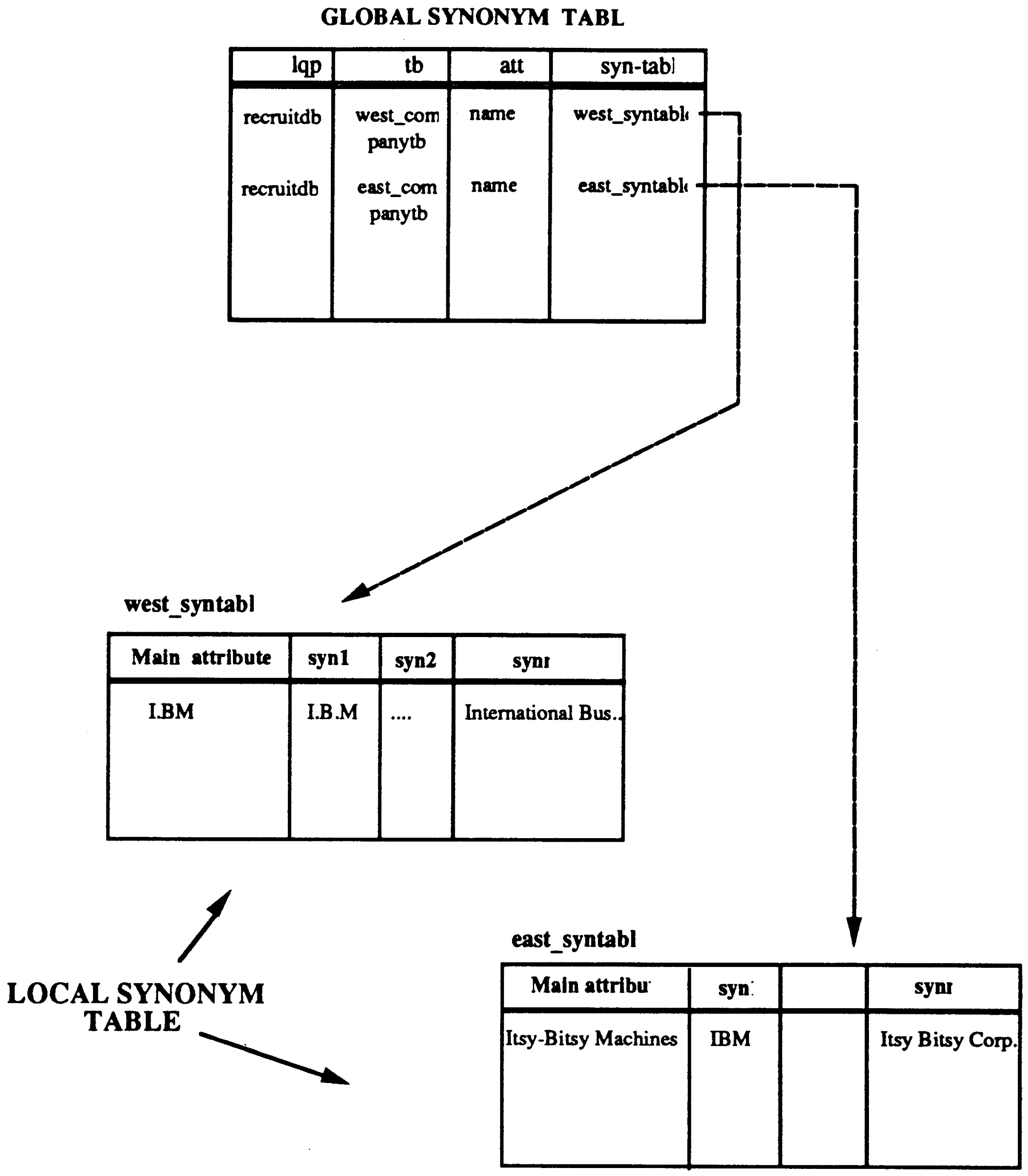


Figure 5.6 Two-Level Scheme for Synonym Catalogs

example, the synonym **west_syntb** contains synonyms for the attribute *(recruitdb west_companytb name).*

5.3 The Global Retrieval Language

The third component of the MERGE Data Model is the language used for querying the global schema. The Global Retrieval Language (GRL) provides a common query language for retrieving and joining data expressed in the global schema. GRL is very simple to undertand and supports retrieval-only capabilities.

5.3.1 GRL Design Issues

The objective of GRL is to provide a common language for querying different database systems. Since the query capabilities of each database system varies widely, the choice of the query capabilities that GRL should provide is an important issue.

Presently, CIS/TK is targeted for decision support applications where retrieving data from separate systems is more common than updates. Global updates is not only a difficult technical issue but is also hard to implement in reality due to the autonomy of the various databases. We thus do not focus on update capabilities.

Some of the databases that MERGE intend to support do not have any manipulation capabilities, for example, Reuters, an on-line financial database is a retrieval-only system. In contrast, database systems like ORACLE SQL not only have retrieval capabilities, but they also have data manipulation capabilities like *max*, *min*, and *group*. In order to provide for a common language that can access disparate systems, we had to make a choice between the functionalities offered.

One choice is for GRL to provide for most types of query capabilties, and when a local database does not have a GRL supported capability, for example *max*, MERGE can provide for a global implementation of the capability. However we decided not to implement any manipulation type capabilities to keep the GRL simple and general. Instead manipulation capabilities will be provided at the AQP level, where the manipulation capabilities can be custom built according to the application.

Having decided on retrieval-type operations, there was still the issue of what kinds of retrieval-type capabilities we should support. A key thing that MERGE intends to support is the merging of data from different sources, thus a join capability was necessary.

Another issue in the design of GRL was in the design of the syntax. In the previous prototype, the query language was very LISP oriented, which was hard to undertsand for most users, but more efficient to process within a LISP environment. For the current version of GRL, we compromised on a SQL-like, LISP-like language. The SQL-like syntax will make GRL more easy to understand. Ultimately, a front-end SQL language could be developed as future work to serve as the common query language.

5.3.2 An Overview of GRL

A typical GRL query and the format which it returns data is shown in Figure 5.8. In the next section, we describe how to use some of the features of GRL.

Selecting an Entity
To select a single entity and its attributes in a global schema, the *select* statement is used. For example, to query the entity *alumni* for the attributes *last-name, first-name , and position* with a condition that the *degree* is equal to "SB 79", the following query is used:

```
(select   alumni   (last-name   first-name position)
               where  (=  degree  "SB 79"))
```

"Find the AT&T company's recruiting dates, positions, and alumni who work for that company."

```
GRL:
 (join  (select  company  (position  date)
              where  (=  name  "AT&T"))
        (select  alumni  (last-name  first-name  degree)
        on  works_for)

Data:
(((company  position)  (company  date)  (alumni  last-name)
  (alumni  first-name)  (alumni  degree))
 ("accountant"  "3 March"  "Hotchkiss"  "George"  "MS 79")
 ("engineer"  "4 March"  "Hotchkiss" "George"  "MS 79")
                                                  ... )
```

Figure 5.8 A Typical Global Query in GRL

The data returned looks like:

```
(((alumni  last-name)  (alumni first-name) (alumni position))
 ("Smith"  "John"  "manager")
 ("Hopkins"  "John"  "physician")
  ... )
```

If all the attributes within an entity are to be selected, then the *-option can be used:

```
(select  alumni  *  where  (=  degree "SB 79"))
```

which is equivalent to the following query:

```
(select  alumni  (last-name  first-name  degree  position)
             where  (=  degree "SB 79"))
```

Complicated conditions can also be expressed within a *select* statement. For example, to find all the alumni who have a degree equal to "SB 79" and is working in the position of "manager", the following query is used:

```
(select  alumni  (last-name  first-name)
             where  (and  (=  degree  "SB 79")
                          (=  position  "manager")))
```

Similarly, an *or* condition can be expressed in a similar fashion.

<u>Joining Entities</u>
To join multiple entities, the *join* statement is used. For example, to join the two entities *alumni* and *company*, we can use the following query:

```
(join  (select  company  (position  date)
               where  (=  name  "AT&T"))
        (select   alumni  (last-name  first-name  degree)
         on  works_for)
```

When there is only one relationship between two entities, the query can be specified without the *on* clause. In addition, the join statement supports multiple nested join statements with the following format:

```
(join  (select  entity1   (att1   ... attn ) where ... )
       (join  (select  entity2  (att1  ... attn )  where ... )
              (join  (select  entity3  (att1  ... attn ) where ... )
                     (... ))))
```

For a more detailed description of the GRL syntax, please refer to Appendix B.1.

6 GLOBAL QUERY PROCESSING

In the last chapter, we presented the data model and its associated components. The Global Query Processor (GQP) is the basic engine for executing a global query, using the components of the data model for attribute mapping and data reconciliation. The GQP is part of the CIS/TK query processing architecture and acts as the interface between the local query processors and the application query processor.

Section 6.1 provides an overview of the GQP architecture, and Section 6.2 addresses some of the main issues in developing the GQP. In Sections 6.3 and 6.4, the two main components of the GQP -- the Query Parser and Query Router are described in further detail.

6.1 Overview of the GQP Architecture

The GQP architecture is divided into two main parts: query parsing and query routing. Figure 6.1 summarizes the main subcomponents in the GQP and their interaction. The partitioning of the GQP reflects the two main tasks that happen during query processing: determining the subtasks that need to be done and executing these subtasks. In addition, by separating the parser from the router, we can in the future change the routing algorithm without requiring modifications to the entire GQP. In the previous prototypes, the router was imbedded in the parser. This scheme made it hard to extend the system. Furthermore, it made the system hard to understand and debug. The partitioned parser-router design offers a better alternative.

The Query Parser
The query parser accepts a global query specified in the GRL syntax. It creates a parse tree that maps out all the subtasks that need to be done to satisfy the query. The parser tree is created through four subcomponents in the parser.

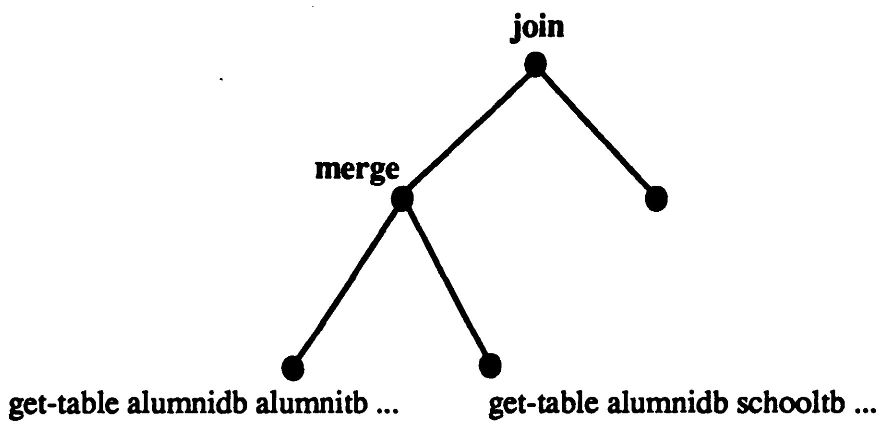

join
merge
get-table alumnidb alumnitb ...
get-table alumnidb schooltb ...

Figure 6.2 Example of Parse Tree

The Query Router
The query router accepts the parse tree. The router is responsible for executing the parse tree and combining the data into a format that reflects the initial global query, for example, removing attributes inserted for joining purposes but not specified in the global query, and converting the local attribute names back into its equivalent global names. The router has four submodules that accomplish the above mentioned tasks.

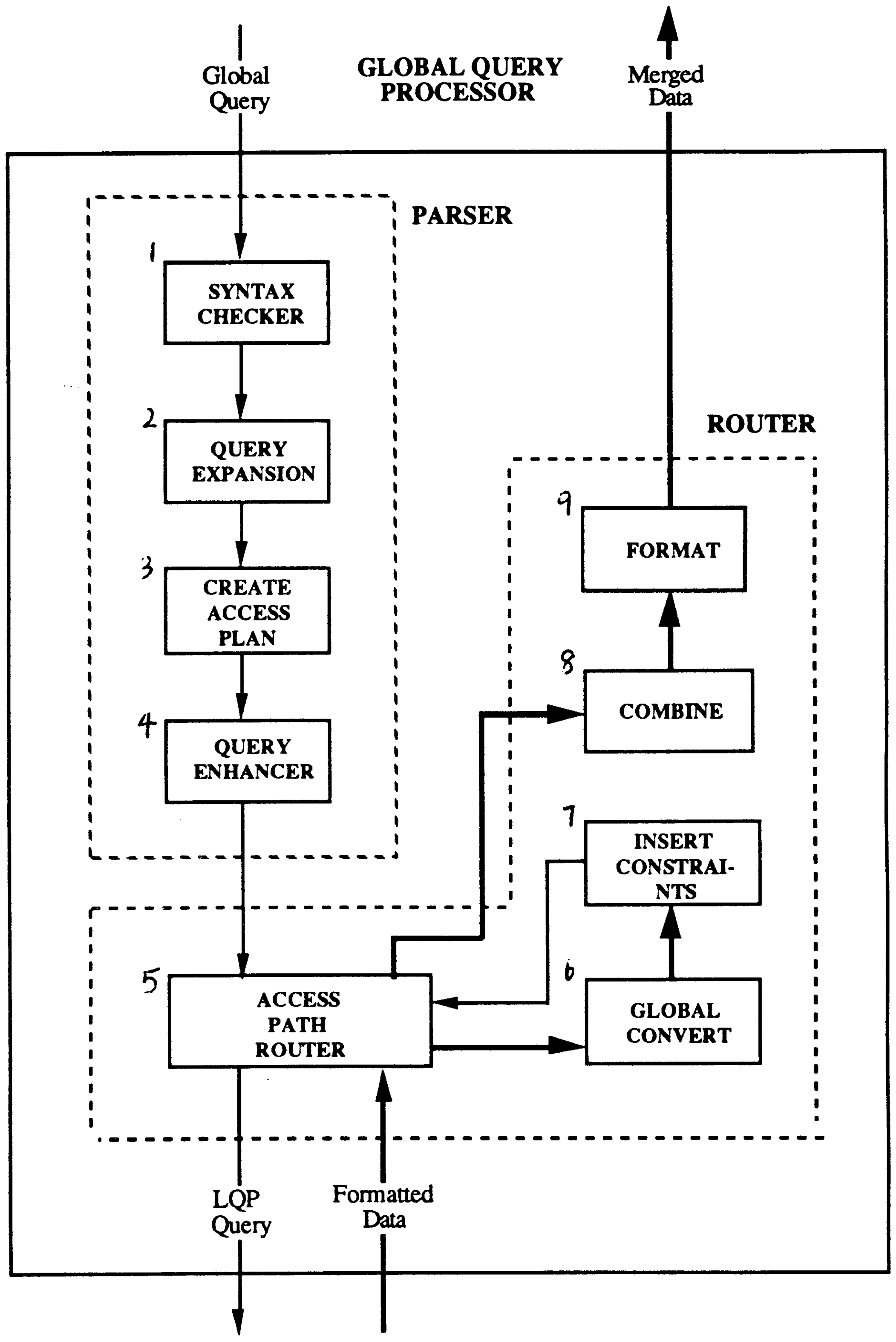


Figure 6.1 The GQP Architecture

The *access plan router* module executes each leaf of the parse tree, and is responsible for invoking the many subqueries to the LQPs. After the execution of each leaf, the data returned is sent to the *global convert* module, which maps the local attribute names into the equivalent global names. The *insert* module then builds a set of constraints that is inserted into the next leaf of the parse tree. The execute-convert-insert loop is completed when the entire parse tree is executed. All the data is then sent to the *combine* module where it is combined. Finally, the combined data is formatted by the *format* module into a form that reflects the initial global query.

The previous section has presented an overall view of the main components of the GQP and their interactions. In the following section, we will present how the GQP tackles some interesting issues posed by query processing in a distributed database environment.

6.2 Issues in Global Query Processing

6.2.1 Automatic Database Selection

In a distibuted database system, data can usually be retrieved from several sources. The problems faced in database selection are mainly due to (1) overlapping data, and (2) replicated data. When the number of underlying databases is large, it is infeasible to expect the user to manually select the databases that correspond to a global query -- some mechanism that aids or automates the selection process is required.

The Problem - Many Combinations To Choose From

Figure 6.3 shows a global query fragment that is mapped to several fragments in the underlying data. For the global attribute *att1*, there are two possible fragments (or sources) where the data can be retrieved, i.e., *d1* or *db2*. For *att2*, the data can be retrieved from either fragments *db3* or *db4*, which are overlapped. However, *att3* can only be retrieved from *db4*.

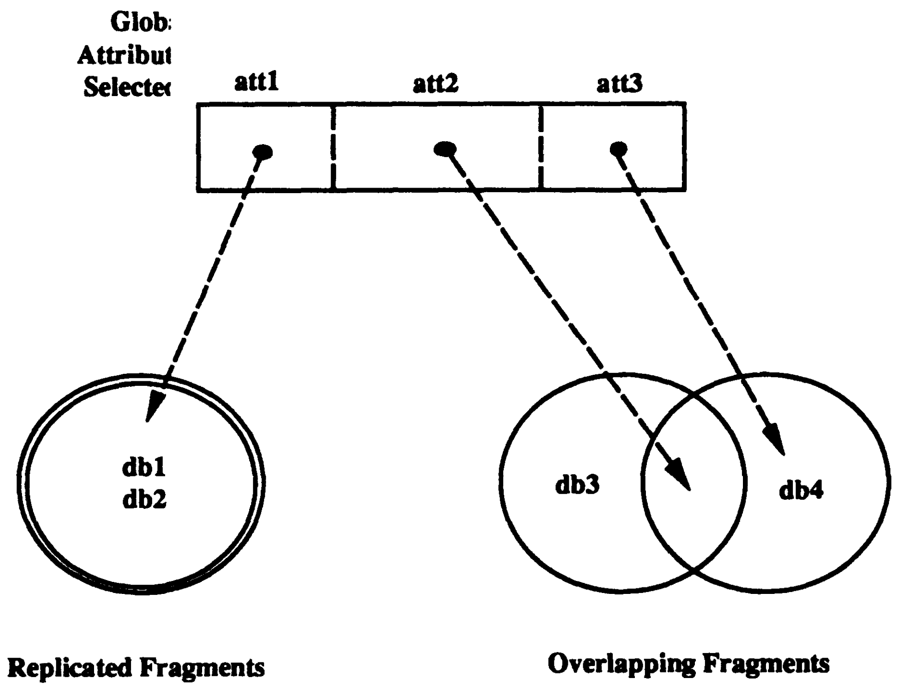


Figure 6.3 Mapping of Global Attributes To Possible Fragments

Thus to satisfy the global query, the possible combination of fragments to select include the following:

1. (db1 db3 db4),
2. (db1 db4),
3. (db2 db3 db4), or
4. (db2 db4)

Faced with several choices, a combination can be selected on a number of possible criterions, for example, on the least number of fragments, on the lowest communication costs or on the least communication time delay. For example, if we want to optimize on the number of sources accessed, we would either choose combinations 2 or 3 since they require access to only two fragments.

A Changeable Set of Selection Rules

Choosing a particular combination of sources is based on factors that are usually dependent on the application and the requirements of the user. For example, in financial applications, knowledge and the ability to choose the source of the data is an important criterion stressed by many users [PAG 89]. In most DDBMS, the selection mechanism is fixed and imbedded within the routing algorithm. In MERGE, we recognise the fact that the criterias for source selection often change and have accordingly developed a selection mechanism that utilizes a set of changeable rules for source selection. In addition, options for both automatic selection, manual selection or a mixture of both are possible.

Currently, we have developed a default set of simple rules to automatically select an access path. It is based on the criteria of accessing the least number of fragments, and if possible within one database, or table. These rules are detailed in section 6.3.3. The current set of rules is intended only to show the feasibility of such a selection mechanism and it ignores factors like communication costs and delays. However, by choosing a rule scheme, we will be able to accomodate future extensions.

6.2.2 Join Strategy

The GQP has to join data from multiple databases. One strategy for joining data is to separately query each database and join the data at the global level. In this strategy, the results from a database query are not used in subsequent queries to other databases. The search space for each query is thus rather large.

The other strategy is to use the results from one database query as constraints for the next subquery. This has the advantage of narrowing the search space in the subqueries.

In our GQP, the second strategy is adopted. The MERGE system is targeted for decision support applications where the amount of data retrieved is usually small but involves several databases. Compared to the second strategy, the first strategy results in large amounts of data being retrieved from each database. This significantly lengthens the total retrieval time.

The retrieval time for the first strategy can be significantly improved if each query can be executed in parallel. However, our present communications server cannot handle multiple tasks. A new communications server that can handle multiple tasks is currently being implemented [GAN 89].

6.2.3 Local DBMS Optimizations

Most DBMS have capabilities for joining and manipulating data. In a distributed database system, a major issue is whether the system should make use of the local DBMS's capabilities. Using the capabilities of local DBMS has the advantage of relieving the global query processor from extra processing.

In our version of the GQP, we chose not to make this local DBMS optimization. The main reason being that such a feature would require a more complex GQP, since Merge is designed to retrieve data from heterogeneous databases with varying capabilties. For example in Multibase, a catalog is used to keep track of the capabilities supported by each database. If a query to Multibase uses a capability that is not found in the local database, it wil augment such a capability at the global level. However, this incurs the cost of extra checks and augmentation, making the global query processor much more complicated. At presently, we do not intend to optimize the GQP to use local DBMS capabilities, although it serves as an interesting piece of future work, especially in applications where speed is more critical.

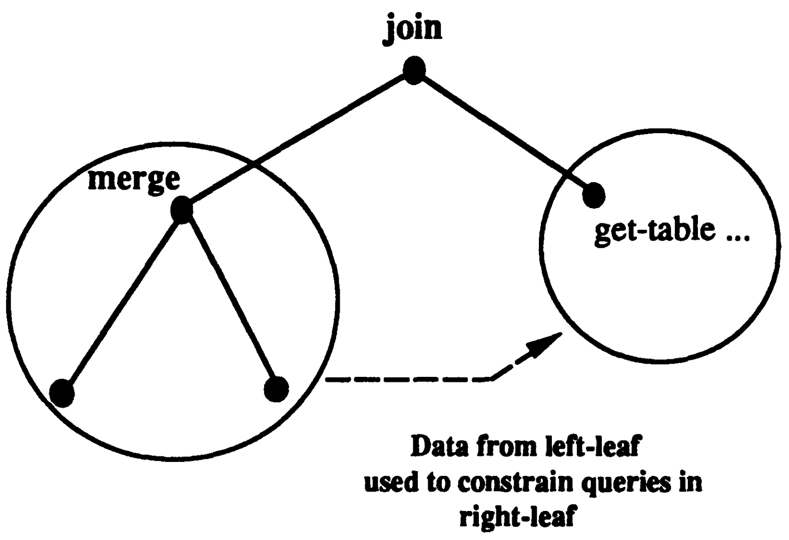


Figure 6.4 GQP Join Strategy

6.2.4 Interfacing for Data Reconciliation

One of the most complex parts of query processing is performing data reconciliation. In Chapter 2, we discussed the needs for reconciling certain types of data conflicts at the GQP level, namely resolving syntax type conflicts so that data from separate sources can be combined. For the reasons of extensibility, data reconciliation in the GQP is actually done by tools that are not imbedded within the GQP. For example, a translation facility [MCC 88] is a tool currently being used in the preliminary version of the GQP for performing translations between different data formats and different scale units. As new tools like instance identification and domain mapping are developed for reconciling data, the GQP should be able to accomodate them. In MERGE, we have developed a consistent interface within GQP for accomodating new tools.

Data reconciliation during query processing can basically happen at two places: (1) before getting the data and (2) after getting the data. For example, consider the following global query:

```
(select company (position date industry) where (= name "AT&T"))
```

(1) Before Getting the Data
The previous query is asking for the "AT&T" company's recruiting positions and dates. However, "AT&T" is also represented as several other names in the underlying data, for example "AT&T Corp.", and "American Telephone + Telegraph". Thus before getting the data, the equivalent synonyms for "AT&T" should be inserted into the query in order to get all "AT&T" company's recuiting information. In the GQP, insertion of synonyms is performed in the *query enhancer* module.

(2) After Getting the Data
After getting the data, the data from different sources have to combined. However, as discussed in Chapter 2, in order to combine the data, the data has first to be resolved for conflicts in naming, formats and

scales. For example, if the previous query retrieved data from two tables, as shown in Figure 6.5, in order to get all the information regarding the "AT&T" company, the company names returned from each database have to be standardized before combining the data returned. Data reconciliation after gettting data is done in the *combine* module of the router.

In the GQP, data reconciliations before getting the data are done in the *query enhancer* module of the parser. Data reconciliations after getting the data are done in the *combine* module of the router. In this way, as new tools are developed to support data reconciliation, there is a consistent way within the GQP to accomodate them. Any other method, like imbedding data reconciliation within the query processor, has the disadvantage of not being easily extendable.

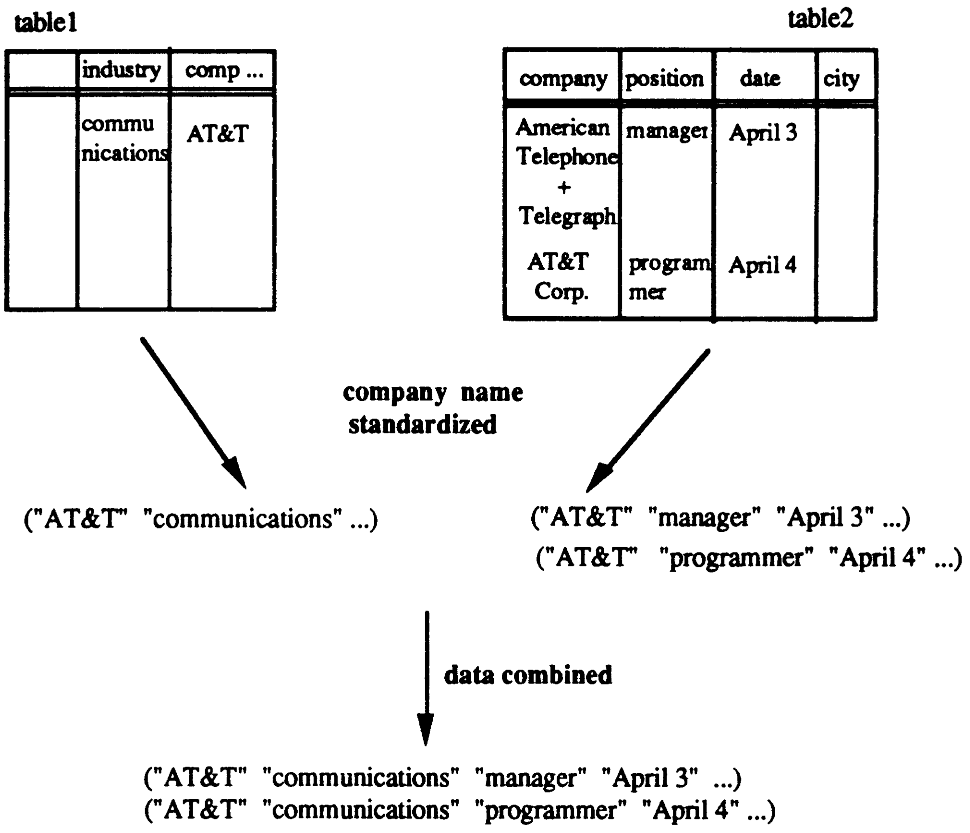


table1
industry
comp ...
commu nications
AT&T
table2
company
position
date
city
American Telephone + Telegraph
manager
April 3
AT&T Corp.
programmer
April 4
company name standardized
("AT&T" "communications" ...)
("AT&T" "manager" "April 3" ...)
("AT&T" "programmer" "April 4" ...)
data combined
("AT&T" "communications" "manager" "April 3" ...)
("AT&T" "communications" "programmer" "April 4" ...)

Figure 6.5 Data Reconciling before Combination

6.3 The Query Parser: How it Works

In this section, we provide a detailed description of how the parser works. Recalling the *simple-placement* global schema described in Chapter 5, a typical GRL query based on that schema is:

"Find the AT&T company's recruiting dates, positions, and alumni who work for that company."

```
(join  (select  company  (position  date)
               where  (=  name  "AT&T"))
         (select   alumni  (last-name  first-name  degree position)
        on   works_for)                                            --- Query (1)
```

This query is accepted by the parser and is transformed into a parse tree. The transformation stages are described next, and they include error checking, query expansion, creating an access plan, and query enhancing.

6.3.1 Stage 1: Error Checking

In the error checking stage, the query is both checked for syntax and lexical errors. Syntax checking involving checking the correctness of the query syntax. In lexical checking, the entities, attributes and relations specified in the query are checked against the current global schema, and an error signalled if an entity, attribute or relation is not found in the global schema.

6.3.2 Stage 2: Query Expansion

In the query expansion stage, the global query is expanded into a form that is easier to manipulate within the GQP. Several types of expansions are involved:

Relation Expansion
First, the join relationship is expanded. The join relationship is the *on* clause of the GRL query. For example in query (1), the join relationship is *works_for*. The join relationship is expanded into the actual join condition. For example, the relationship *works_for* would be expanded into:

```
(=  (company  name)  (alumni  company) )
```

This join information is obtained from the global schema. For our example, this would be the *:join* slot of the*works_for* relation object.

* Expansion
Secondly, the * option is expanded. The * option is used to select all the attributes in an entity. For example, to get all the attributes within the *alumni* entity, the following query can be used:

```
( select   alumni   *   where (= name "Sam"))
```

which is expanded into:

```
( select alumni   (name social_security degree major position company)
                     where ( =  name  "Sam" ) )
```

Join-Key Expansion
Thirdly, the attributes are expanded to include the join-key attributes. For example, we found earlier that query (1) has the join condition:

```
(=  (company  name)  (alumni  company) )
```

The join-key attributes are (company name) and (alumni company), i.e., these two attributes are used these entities. However, query (1) does not specify either of these join-key attributes. A join cannot be performed if data for that attribute is not retrieved. The expanded query for query (1) is:

```
(join (select company (position date name)
              where (= name "AT&T"))
      (select alumni (last-name first-name degree position company)
      on (= (company name) (alumni company))                    --- Query (1.2)
```

Attribute Expansion
The last step in the expansion is to expand each attribute in a GRL statement into a form that is more easier to manipulate. Each attribute is expanded into a list (*entity attribute*).

After query expansion, query (1.2) looks like the following:

```
(join (select company ((company position) (company date) (company name))
              where (= (company name) "AT&T"))
       (select alumni ((alumni last-name) (alumni first-name)
                      (alumni degree) (alumni position) (alumni company)))
       on (= (company name) (alumni company)) )                 --- Query (1.3)
```

Next, the expanded query is passed to the *create access plan* stage.

6.3.3 Stage 3: Creating an Access Plan

In this stage, an access plan is created that maps out all the subtasks that need to be done to satisfy the query. Creating an access plan involves (1) find all possible access paths, and (2) selecting an access path, and (3) creating an access plan (parse tree) based on (2). These steps are summarized in Figure 6.6, and are further elaborated next.

(1) Find Access Paths
To find the access paths for a query, each *select* statement of a query is applied the procedure described next. For our examples, we will use the first *select* statement of query (1.3).

Procedure:
(i) Map Global Attributes to Local Names. Each global attribute is mapped to all the possible local names. For example, the attribute (*company name*), is mapped to the following local names:

```
((recruitdb west_companytb company)
 (recruitdb east_companytb comp_name))
```

After all the global attributes have been mapped into the local names, this map information is stored in a local cache to facilitate quick lookups.

(ii) Joins between Sources. To find the possible access paths, all joins between the sources have to be first enumerated. All the join relationships between the sources can be obtained from the global schema, from the *:table-relations* of the entity object. These relationships are then used to find all possible source combinations. For example, in order to satisfy query (1.3), the sources found previously in (i) which include:

For the company entity:
1.1 (recruitdb west_companytbl)
1.2 (recruitdb east_companytbl), and

For the alumni entity:
2.1 (alumnidb alumnitb)
2.2 (alumnidb schooltb)

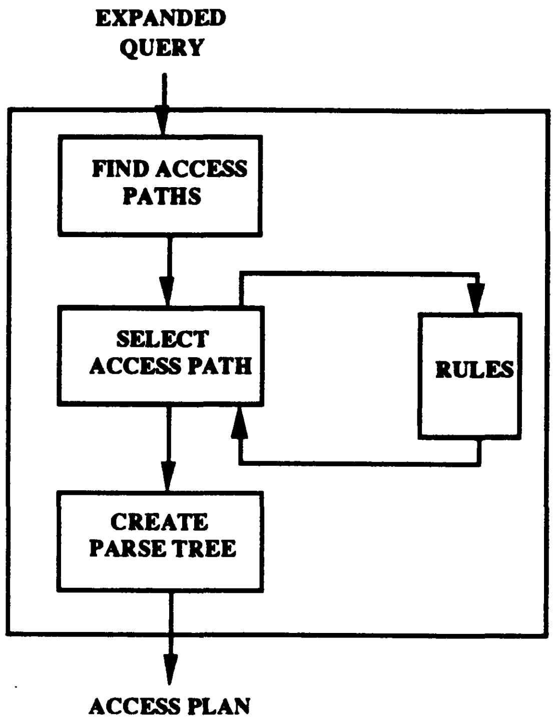


Figure 6.6 Creating an Access Plan

have relations of a *concatenate* and *merge* respectively. In other words, in order to satisfy the global query that involves the entity *company*, the two sources 1.1 and 1.2 need to be concatenated together. Similarly, to satfisfy the global query for the *alumni* entity, the two sources 2.1 and 2.2 need to be merged.

<u>Step 2: Select Access Path</u>
The selection of an access path is by default done automatically. The default rule set is shown in Figure 6.7. The goal of the default rule set is to determine the least number of sources needed to satisfy a query.

In our example query (1.3), the selection rule applied is very simple because there is only one combination of sources required to satisfy the query, that is, the *only combination?* near the top of the flow chart in Figure 6.6 is found to be true, and the process ends.

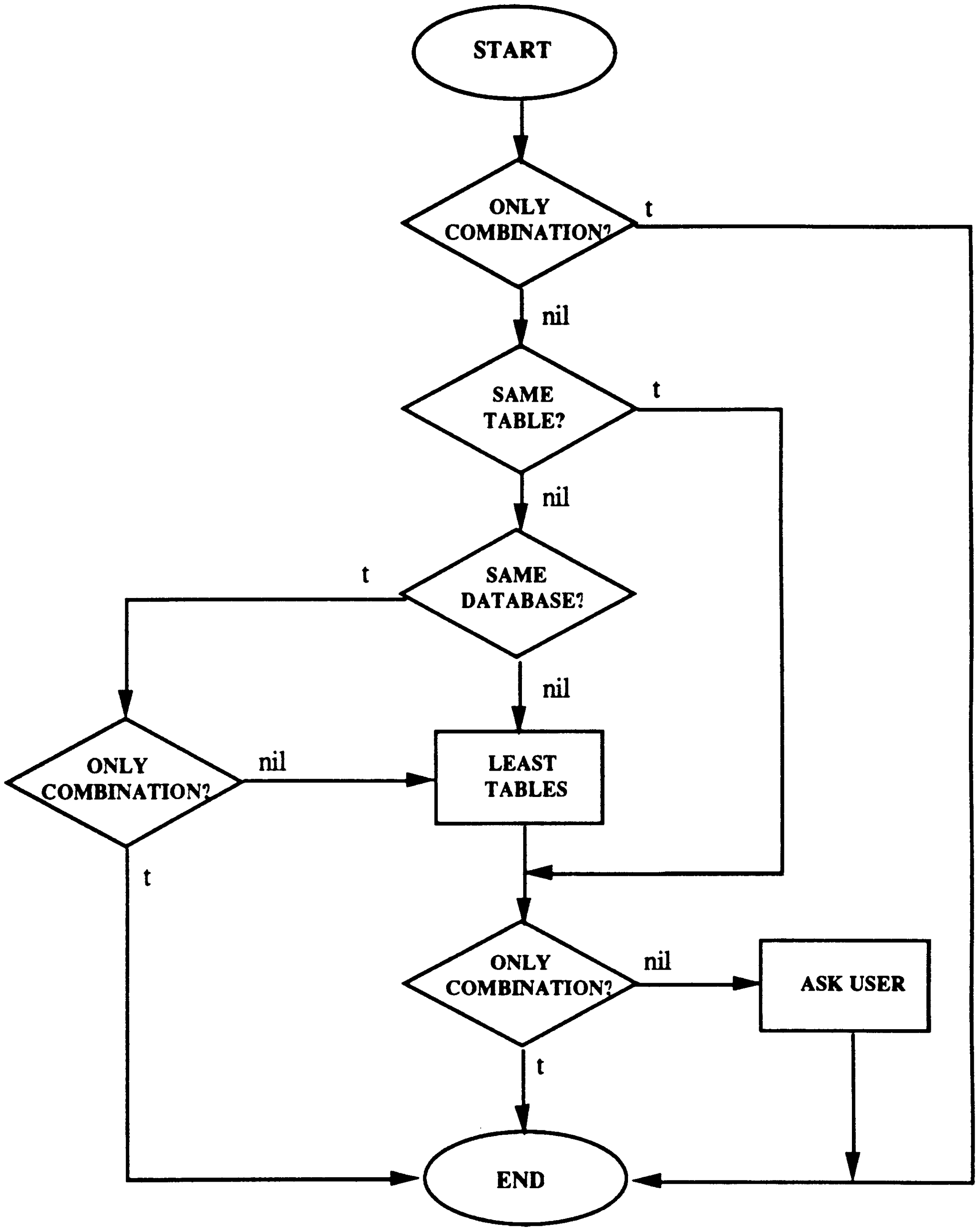


Figure 6.7 Default Selection Rules

Step 3: Create a Parse Tree

The last step is to create the parse tree using the access path selected from step (2). The parse tree created for query 1.3 is shown in Figure 6.8. The parse tree is specified in an intermediate query language for which the router understands.

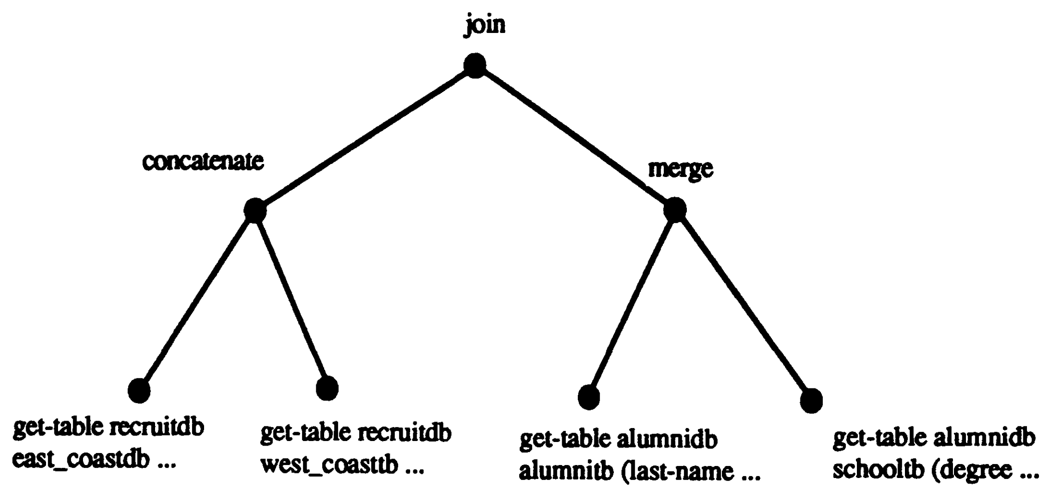

join
concatenate
merge
get-table recruitdb east_coastdb ...
get-table recruitdb west_coasttb ...
get-table alumnidb alumnitb (last-name ...
get-table alumnidb schooltb (degree ...

Figure 6.8 Parse Tree for Query 1.3

The parse tree for query (1.3) is the following intermediate query:

```
(join (concatenate (get-table recruitdb west_companytb (position date company)
                              where (= company "AT&T"))
                    (get-table recruitdb east_companytb (position date comp_name)
                              where (= comp_name "AT&T"))
      (merge (get-table alumnidb alumnitb (last-name first-name position
                                                      company ss))
             (get-table alumnidb schooltb (degree ss)
             on (= (alumnidb alumnitb ss)
                   (alumnidb schooltb ss))
      on (= (company name) (alumni company)))
```

This parse tree is then sent to the query enhancement stage.

6.3.4 Stage 4: Query Enhancing

The two types of query enhancement include synonym identification and translations. These enhancements are described next.

Synonym Identification

The first type of query enhancement is synonym identification. All attributes in a query are checked against the synonym catalog for synonyms. For example, to check whether *(recruitdb west_companytb company)* has synonyms, the following command is used:

```
(get_syntb 'recruitdb  'west_companytb 'company)
```

If synonyms exist, the local synonym table for that attribute is returned, else nothing is returned. Recall our example in Figure 5.5 from Section 5.2.1 on the two-level scheme representation for synonym catalogs. The local synonym table for *(recruitdb west_companytb company)* from that example would be:

```
*west_syntb*
```

Each synonym table is inserted into the parse tree at the leaf (get-table statement) where the synonym occurred in the following format:

```
(get-table  lqp tb (att1 ... attn) where conds
                syns ((att1  syn_table1)  ...  (attn   syn_tablen))
```

where *attn* is the name of the local attribute that corresponds to the synonym table *syn_tablen*. For example **west_syntb** would be inserted as:

```
(get-table   recruitdb   west_companytb   (position   date name)
                              where  (=  name  "AT&T")
                               syns   ((name   *west_syntb*)))
```

After all the attributes are checked, the parse tree is augmented to include these synonym table names. The actual insertion of synonyms does not take place until query routing.

After query enhancements, the parse tree for query (1.3) is the following:

```
(join (concatenate (get-table recruitdb west_companytb (position date company)
                              where (= company "AT&T")
                               syns ((company *west_syntb*)))
                   (get-table recruitdb east_companytb (position date comp_name)
                              where (= comp_name "AT&T")
                              syns ((comp_name *east_syntb*)))
   (merge (get-table alumnidb alumnitb (last-name first-name position
                                                  company ss))
          (get-table alumnidb schooltb (degree ss)
          on (= (alumnidb alumnitb ss)
                (alumnidb schooltb ss))
    on (= (company name) (alumni company)))                --- Parse Tree (1.3)
```

6.4 Query Router: How it Works

The query router accepts a parse tree which it then executes. Before going into the details of how each module of the router works, we will run through an example using the parse tree created for query (1.3). From hereon, we will refer to that parse tree as parse tree (1.3). The numbers in bold in the following example correspond to where the parse tree are being processed within the router, as shown in Figure 6.1. For convenience, we reproduce parse tree (1.3):

ACCEPTS:

```
(join (concatenate (get-table recruitdb west_companytb (position date company)
                                  where (= company "AT&T")
                                  syns ((company *west_syntb*)))
                   (get-table recruitdb east_companytb (position date comp_name)
                                  where (= comp_name "AT&T")
                                  syns ((comp_name *east_syntb*)))
      (merge (get-table alumnidb alumnitb (last-name first-name position
                                           company ss))
             (get-table alumnidb schooltb (degree ss)
             on (= (alumnidb alumnitb ss)
                   (alumnidb schooltb ss))
       on (= (company name) (alumni company)))
```

The router traverses the parse tree in a left to right, depth-first mode. For parse tree (1.3), the first left branch:

5(a):

```
(concatenate (get-table recruitdb west_companytb (position date company)
                                  where (= name "AT&T")
                                  syns ((company *west_syntb*)))
             (get-table recruitdb east_companytb (position date comp_name)
                                  where (= comp_name "AT&T")
                                  syns ((comp_name *east_syntb*)))
```

would be first executed. The *access path* module executes this branch by generating subqueries to the appropriate LQPs. The data returned from the LQPs is combined:

6(a):

```
(((recruitdb west_companytb position) (recruitdb west_companytb date)
  (recruitdb west_companytb company))
 ("manager" "February 5" "AT&T")
  ...
 ("programmer" "February 6" "AT&T"))
```

This data is sent to the *global convert* module which converts the header list (the first list in the data) into the equivalent global attribute names:

7(a):

```
(((company position) (company date) (company name)
  ("manager" "February 5" "AT&T")
  ...
  ("programmer" "February 6" "AT&T"))
```

This is processed by the *insert constraints* module which takes the data and builds constraints for the right branch of parse tree (1.3). These constraints are inserted into the right branch:

5(b):

```
(merge (get-table alumnidb alumnitb (last-name first-name position
                                      company ss)
                      where (=  company  "AT&T"))              ;; constraint inserted
       (get-table alumnidb schooltb (degree ss)
        on (= (alumnidb alumnitb ss)
              (alumnidb schooltb ss))
```

This right branch of parse tree (1.3) is then executed by the *access router* module. The data returned from the LQPs are combined and sent to the *global convert* module:

6(b):

```
(((alumnidb alumnitb last-name) (alumnidb alumnitb first-name)
  (alumnidb alumnitb position) (alumnidb alumnitb company) (alumnidb alumnitb ss))
 ("Ernest" "George" "accountant" "AT&T" "888002147")
 ...
 ("Horton" "Dave" "engineer" "AT&T" "214700888"))
```

The converted data is sent to the *insert constraints* module:

7(b):

```
(((alumni last-name) (alumni first-name) (alumni position)
  (alumni company) (alumni ss))
  ("Ernest" "George" "accountant" "AT&T" "888002147")
  ...
 ("Horton" "Dave" "engineer" "AT&T" "214700888"))
```

However, no constraints are built because all the branches of the parse tree have been executed. The next stage involves combining all the data returned from the left and right branches:

8:

```
(join ((company position) (company date) (company name)
       ("manager" "February 5" "AT&T")
         ...
       ("programmer" "February 6" "AT&T"))
       (((alumni last-name) (alumni first-name) (alumni position)
          (alumni company) (alumni ss))
        ("Ernest" "George" "accountant" "AT&T" "888002147")
          ...
       ("Horton" "Dave" "engineer" "AT&T" "214700888"))
        on (= (company name) (alumni company)))
```

This data is joined into one big list:

9:

```
(((company position) (company date) (company name)
  (alumni last-name) (alumni first-name) (alumni position)
  (alumni company) (alumni ss))
 ("manager" "February 5" "AT&T" "Ernest" "George" "accountant" "AT&T"
       "888002147" )
   ...
 ("programmer" "February 6" "AT&T" "Horton" "Dave" "engineer" "AT&T"        "214700888")))
```

This is processed by the *format* module which removes any attributes not specified in the original query. Refering to the orginal query (1.3), this includes removing *(alumni company)* and *(alunmi ss)*, which were necessary in joining the data but not specified in query (1.3):

RETURNS:

```
(((company position) (company date) (company name)
  (alumni last-name) (alumni first-name) (alumni position)
 ("manager" "February 5" "AT&T" "Ernest" "George" "accountant")
   ...
 ("programmer" "February 6" "AT&T" "Horton" "Dave" "engineer" ))
```

This section has provided a run-through of how the modules in the router interact. In the next section, we describe how each module works.

6.3.1 The Access Path Router

The intermediate query router recognizes four operators, which in its basic form are the following:

GET-TABLE *lqp table (att1 ... attn)* **WHERE** *conds*. Selects the attributes *att1,.. attn* from the table *table* on the restriction *conds.*

MERGE *get-table get-table* **ON** *conditions.* Merges two sets of data returned from the *get-table* statements using the *conditions* as restrictions. All duplicate entries in the data are eliminated.

CONCATENATE *get-table get-table.* Concatenates two sets of data returned from the *get-table* statements. Does not eliminate any duplicates.

The *access path router* accepts a parse tree which it then proceeds in a left-to-right depth -first manner to break down into intermediate queries. The intermediate quries are then executed. When the *access path router* encounters a *get-table* statement, the appropriate LQP specified in the statement is invoked in the following manner:

(send-message *lqp* :get-data *(table (att1 ... attn))* *conds*)

After executing all the *get-table* statements within a subquery, the data returned from the LQPs are combined with the either the *merge* or *concatenate* operator.

6.3.2 Global Convert

The global convert accepts a list of data from the access path router and converts the header of that list into the equivalent global names. For example in 6(a), the header list is:

```
((recruitdb west_companytb position) (recruitdb west_companytb date)
 (recruitdb west_companytb company))
```

Each of these local attributes is converted into its equivalent global attributes by looking up in a temporary cache, created during the parsing of the query. For example, to look up the global attribute for the first local attribute in the header list shown above, the following command is used:

```
(lookup-3map 'recruitdb 'west_companytb 'position *loc->gs*)
```

where **loc->gs** is the name of the local cache. The local attribute name returned is:

```
((company position))
```

This is done for all the elements in the header list, which is then appended to the rest of the data into the following list:

```
(((alumni  last-name)  (alumni first-name)  (alumni  position)
   (alumni company)  (alumni  ss))
  ("Ernest"  "George"  "accountant"  "AT&T"  "888002147")
  ...
 ("Horton"  "Dave"  "engineer"  "AT&T"  "214700888"))
```

6.3.3 Insert Constraints

This module takes the data returned from one branch of the parse tree and uses it to constrain the next query found in the right branch. For example in 7(a), the data returned from the left branch of parse tree (1.3):

```
(((company  position)  (company  date)  (company  name)
  ("manager"  "February 5"  "AT&T")
   ...
  ("programmer"  "February 6"  "AT&T"))
```

is matched with the *on* part of the *join* statement, the condition being:

```
(=  (company  name)  (alumni  company))
```

A match is found when one of the attributes in the header list of the data match with an attribute in the condition list. In this case, a match is found for *(alumni company)*. The data for the match is found from the data and used as constraints:

```
(=  (alumni company)  "AT&T")
```

which is converted into the local attribute name:

```
(=  (alumniidb  alumnitb  company)  "AT&T")
```

and inserted into the left-most leaf in the right branch of the parse tree:

```
(merge (get-table alumnidb alumnitb (last-name first-name position
                                          company ss)
                   where (=  company  "AT&T"))              ;; constraint inserted
        (get-table alumnidb schooltb (degree ss)
        on (= (alumnidb alumnitb ss)
              (alumnidb schooltb ss))
```

6.3.4 Combine

The combine module takes the data returned from each branch of the parse tree and combines it on the *join* operator. The joining process involves a cartesian product of the data and then a restriction is performed on the resulting data list.

In the future, when data reconciliation facilities like translations are implemented, they can be interfaced to the GQP in the *combine* module.

6.3.5 Format

The format module takes the combined data and strips off attributes that were not specified in the initial query but were used in the the joining process. The data is then returned to the caller of the GQP, completing the query processing process.

The last two chapters described the data model and the global query processor. In the next chapter, we test these components with an application called the Placement Assistant System.

7 APPLICATION: PLACEMENT ASSISTANT SYSTEM

This chapter describes a simplified version of the Placement Assistant System (PAS) being implemented by the CIS/TK project. It is used to demonstrate the MERGE system operating within the CIS/TK environment, which currently supports access to several SQL-based DBMS. In the next section, we describe the operational scenario of the simplified PAS, and in section 7.2, show a sample session with the system.

7.1 Implementation Scenario

The following describes the scenario of the PAS system:

As a student, it would be nice to have a Placement Assistant System (PAS) to help plan and prepare you for your job interviews. This task normally involves selecting a set of companies on any several criteria, such as industry, location, economic performance, position. You will then want to check which companies will be sending recruiters to your school, resolve any conflicts, and define your schedule of interviews. In order to focus your energies and improve your chances, you will want to gather relevant information from

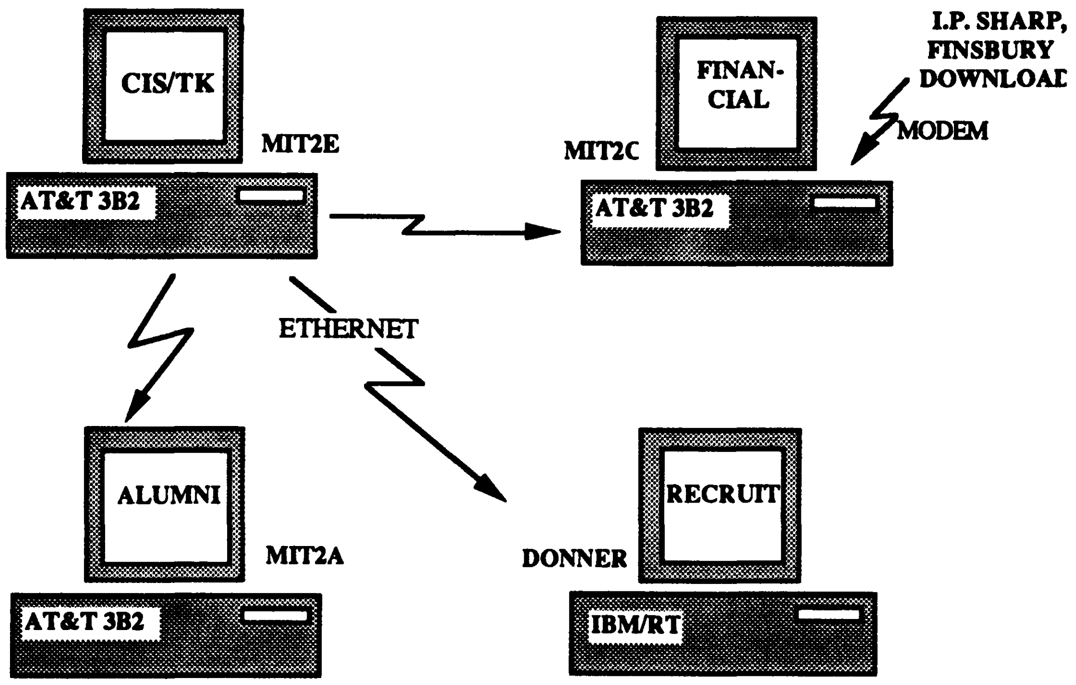


Figure 7.1 Machine Configuration for PAS

both external and internal sources (if it happens that an alumnus works for any of the companies). This would allow you to be knowledgeable about the company, prepare you to ask questions, and solicit support for your application.

The Placement Assistant System is to be an on-line system that helps you in the various phases of the placement process. There are several databases, shown in Figure 7.1, at your disposal:

1- **ALUMNI** (on an AT&T 3B2 computer). This will give you access to data regarding alumni and the corporations which employ them,

2- **RECRUIT** (on an IBM PC/RT computer). The RECRUIT database, maintained by the Placement Office at SLOAN, provides information as to which companies are recruiting, the positions for which they are hiring, and when they will be coming.

3- **FINSBURY** and **I.P. SHARP** (external databases). Commercial data banks such as Finsbury or I.P. Sharp provide general information about location, industry, products, financial situation of major corporations.

Presently, this version of PAS does not have the capability to access the external databases through CIS/TK, so the data from the external databases is downloaded onto an SQL database (Financial on MIT2C) which is then accessed by the CIS/TK system. Efforts to provide on-line connection to the external databases are near completion and are further described in [GER 89] [GAN 89].

In the next section, we describe a sample session with MERGE.

7.2 Sample Session

MERGE provides a common query language for retrieving, and combining data from the various databases described in the last section. A global schema that represents the underlying data is shown in Figures 7.2(a) and 7.2(b). The following is a session that a student might go through with MERGE to find out more about recruiting companies:

1. **"Find all companies recruiting in the communications industry".** This query involves accessing two databases (alumni and recruit). The first access is to the alumni database, which gets the Standard Industry Code (SIC) for the communications industry, and then the recruit database is accessed to get companies with that SIC.

;;;; Query to Global Query Processor:

```
(GQP  (SELECT  COMPANY  (DATE  POSITION  NAME)
                 WHERE  (=  INDUSTRY  "Communications")))
```

;;;; This query is sent to the parser which returns the following parse tree:

```
<2  (PARSER  (MERGE  (GET-TABLE  LOCAL2E  SICCODETB
                                 (INDUSTRY  SIC_CODE)  WHERE
                                       (=  INDUSTRY  "Communications"))
                          (GET-TABLE  ORACLE2E  COMPANYTBL
                           (VISIT_DAY  POSITION  COMPANY_NAME
                                                 SIC_CODE))
                      ON
                          (=  (ORACLE2E  COMPANYTBL  SIC_CODE)
                              (LOCAL2E  SICCODETB  SIC_CODE)))
                   . . .
```

;;;; The parse tree is then passed to the router which routes each subquery to the appropriate LQP:

```
2>  (QUERY_ROUTER
        (MERGE  (GET-TABLE  LOCAL2E  SICCODETB  (INDUSTRY  SIC_CODE)
                                      WHERE  (= INDUSTRY  "Communications"))
                (GET-TABLE  ORACLE2E  COMPANYTBL
                                     (VISIT_DAY  POSITION  COMPANY_NAME
                                                 SIC_CODE))
              ON
                (=  (ORACLE2E  COMPANYTBL  SIC_CODE)
                    (LOCAL2E  SICCODETB  SIC_CODE))))
```

Global Schema:

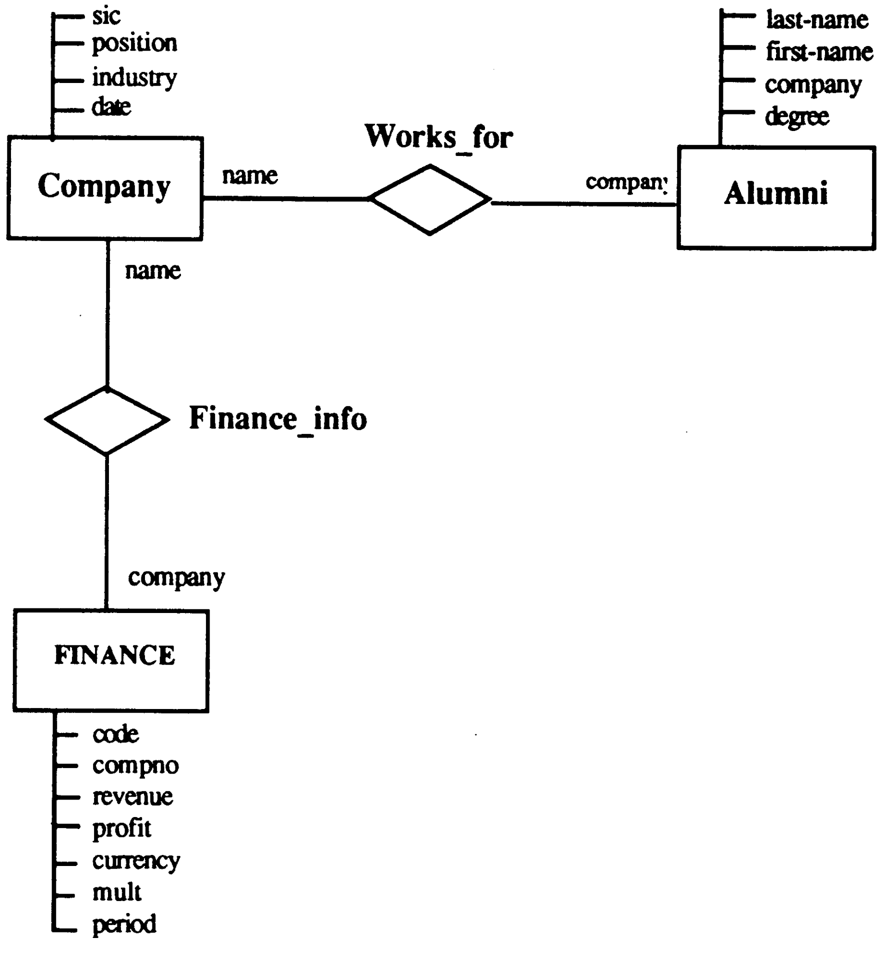


Figure 7.2(a) Global Schema for PAS

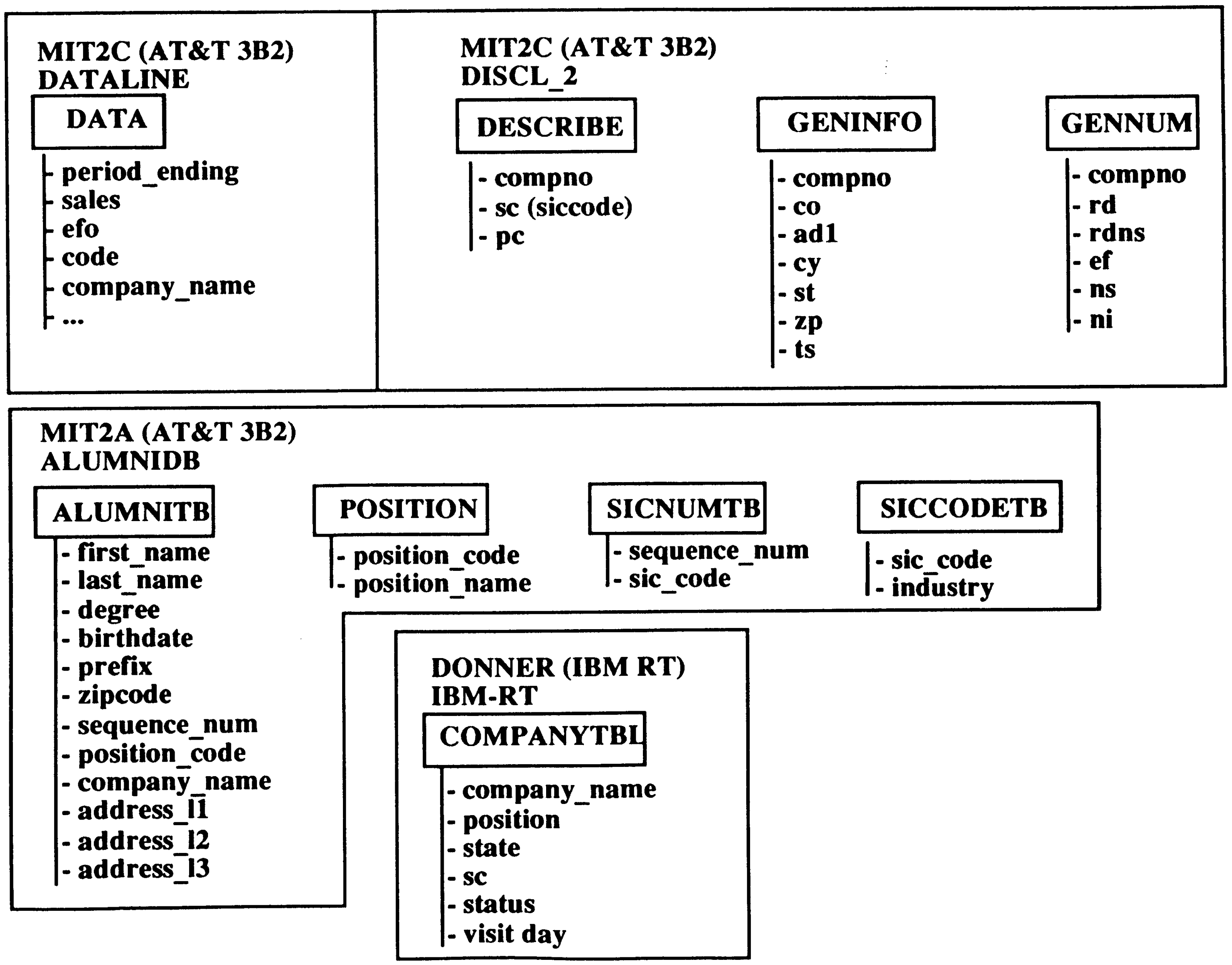


Figure 7.2(b) Underlying Tables for PAS

;;;; The first subquery is to the alumni database to get the SIC for "communications":

```
3>  (SEND-MESSAGE  LOCAL2E  :GET-DATA
                  (SICCODETB  (INDUSTRY  SIC_CODE)
                       (= INDUSTRY "Communications")))

SQL query to be sent to DBMS....
SELECT INDUSTRY, SIC_CODE FROM SICCODETB WHERE INDUSTRY =
'Communications'

Connecting to localdb on machine mit2e...Done.
```

;;;; The LQP returns the following data:

```
<3  (SEND-MESSAGE
          (("INDUSTRY"  "SIC_CODE")
            ("Communications"  "48")))
```

;;;; Next, the router executes the right branch (recruiting information) with the newly found information on SIC as a constraint:

```
3>  (QUERY_ROUTER
      (GET-TABLE  ORACLE2E  COMPANYTBL
           (VISIT_DAY  POSITION  COMPANY_NAME  SIC_CODE)  WHERE
                    (= SIC_CODE "48")))
```

;;;; Get data from about recruiting information:

```
4>  (SEND-MESSAGE  ORACLE2E  :GET-DATA
      (COMPANYTBL  (VISIT_DAY  POSITION  COMPANY_NAME  SIC_CODE)
                     (= SIC_CODE "48")))

SQL query to be sent to DBMS....
SELECT VISIT_DAY, POSITION, COMPANY_NAME, SIC_CODE FROM
COMPANYTBL WHERE SIC_CODE = '48'

Connecting to oracldb on machine mit2e...Done.

<4  (SEND-MESSAGE
        (("VISIT_DAY"  "POSITION"  "COMPANY_NAME"  "SIC_CODE")
          ("February 5"  "investment mgmt"  "AT&T"  "48")
          ("January 28"  "finance"  "AT&T"  "48")
          ("January 29"  "marketing"  "AT&T"  "48")
          ("February 9"  "international"  "AT&T"  "48")))
```

;;;; The data is combined with the previous data and returned:

```
<1  (GQP  (((COMPANY DATE)  (COMPANY POSITION)  (COMPANY NAME))
               ("February 5"  "investment mgmt"  "AT&T")
              ("January 28"  "finance"  "AT&T")
              ("January 29"  "marketing"  "AT&T")
               ("February 9"  "international"  "AT&T")))
```

2. **"Find the alumni who work at AT&T, and the company's financial information for the year 1987".** This query involves access to three databases: the alumni, recruit, and financial databases. First the alumni database is accessed to retrieve data about the alumni, and then the recruit database is accessed to retrieve the states. Finally, the IPSHARP database is accessed to retrieve AT&T's financial data for the year 1987. This data is then combines together.

;;;; Query to Global Query Processor:

```
1>  (GQP  (JOIN  (SELECT  ALUMNI  (LAST-NAME  FIRST-NAME  DEGREE)
                        WHERE  (=  COMPANY  "AT&T"))
                    (JOIN  (SELECT  COMPANY  (STATE))
                            (SELECT  FINANCE  (PROFIT  CURRENCY  MULT)
                              WHERE    (=  PERIOD  "19871231")))))
```

;;;; This query is sent to the parser which returns:

```
<2  (PARSER  (JOIN  (GET-TABLE  LOCAL2E  ALUMNITB
                                (COMPANY_NAME  LAST_NAME  FIRST_NAME
                                                          DEGREE)
                               WHERE  (=  COMPANY_NAME  "AT&T"))
                     (JOIN  (GET-TABLE  ORACLE2E  COMPANYTBL
                                      (COMPANY_NAME  STATE))
                             (MERGE  (GET-TABLE  DISCLOSURE2E  GENINFO
                                               (CO  CURR  COMPNO))
                                      (GET-TABLE  DISCLOSURE2E  GENNUM
                                               (CF  NS  MULT  COMPNO)
                                         WHERE    (=  CF  "19871231"))
                                   ON
                                     (=  (DISCLOSURE2E  GENINFO
                                                          COMPNO)
                                         (DISCLOSURE2E  GENNUM
                                                          COMPNO)))
                           ON  (=  (COMPANY  NAME)
                                   (FINANCE  COMPANY)))
                      ON  (=  (ALUMNI  COMPANY)  (COMPANY  NAME)))
```

;;;; This parse tree is sent to the router:

```
2>  (QUERY_ROUTER  (GET-TABLE  LOCAL2E  ALUMNITB
                         (COMPANY_NAME  LAST_NAME  FIRST_NAME  DEGREE)
                           WHERE  (=  COMPANY_NAME  "AT&T")))
```

;;;; invoke LQP

```
3>  (SEND-MESSAGE  LOCAL2E  :GET-DATA
                    (ALUMNITB  (COMPANY_NAME  LAST_NAME  FIRST_NAME
                                                          DEGREE)
                          (=  COMPANY_NAME  "AT&T")))

SQL  query  to  be  sent  to  DBMS....
SELECT  COMPANY_NAME,  LAST_NAME,  FIRST_NAME,  DEGREE  FROM
ALUMNITB  WHERE  COMPANY_NAME  =  'AT&T'
```

```
Connecting to localdb on machine mit2e...Done.
```

;;;; data returned from LQP

```
<3   (SEND-MESSAGE
                   (("COMPANY_NAME" "LAST_NAME"
                             "FIRST_NAME""DEGREE")
                    ("AT&T" "George" "Ernest" "SM 1979")))
```

;;;; routes next leaf in parse tree, which gets the state information

```
2>   (QUERY_ROUTER
                (GET-TABLE ORACLE2E COMPANYTBL
                      (COMPANY_NAME  STATE)  WHERE
                              (=  COMPANY_NAME  "AT&T")))
```

;;;; invokes the lqp for the recruiting database

```
3>   (SEND-MESSAGE  ORACLE2E  :GET-DATA
               (COMPANYTBL (COMPANY_NAME STATE)
                           (=  COMPANY_NAME  "AT&T")))
```

;;;; which returns

```
<3   (SEND-MESSAGE
                   (("COMPANY_NAME" "STATE") ("AT&T" "MA")
                   ("AT&T" "NJ") ("AT&T" "MA") ("AT&T" "MA")))
```

;;;; routes next leaf

```
2>   (QUERY_ROUTER
                (MERGE (GET-TABLE DISCLOSURE2E GENINFO (CO CURR
                                                          COMPNO)
                          WHERE (= CO "AT&T"))
                       (GET-TABLE DISCLOSURE2E GENNUM
                           (CF NS MULT COMPNO)
                          WHERE (= CF "19871231"))
                   ON
                       (= (DISCLOSURE2E GENINFO COMPNO)
                          (DISCLOSURE2E GENNUM COMPNO))))
```

;;; invokes LQP

```
3>   (SEND-MESSAGE  DISCLOSURE2E  :GET-DATA
                   (GENINFO (CO CURR COMPNO) (= CO "AT&T")))

SQL query to be sent to DBMS....
SELECT CO, CURR, COMPNO FROM GENINFO WHERE CO = 'AT&T'

Connecting to discl_2 on machine mit2e...Done.
```

;;;; data returned

```
<3  (SEND-MESSAGE  (("CO"  "CURR"  "COMPNO")  ("AT&T"  "$-US"
                                                              "470")))
```

;;;; route next last leaf

```
3>  (QUERY_ROUTER
                (GET-TABLE DISCLOSURE2E GENNUM
                     (CF  NS  MULT  COMPNO)  WHERE
                     (AND  (= CF  "19871231")  (= COMPNO  "470"))))
```

;;;; invokes the LQP for financial data

```
4>  (SEND-MESSAGE  DISCLOSURE2E  :GET-DATA
                   (GENNUM  (CF NS MULT COMPNO)
                            (AND  (= CF  "19871231")
                                  (= COMPNO  "470"))))

SQL query to be sent to DBMS....
SELECT CF, NS, MULT, COMPNO FROM GENNUM
WHERE (CF = '19871231') AND (COMPNO = '470')

Connecting to discl_2 on machine mit2e...Done.
```

;;;; data returned by LQP

```
<4  (SEND-MESSAGE
                   (("CF"  "NS"  "MULT"  "COMPNO")
```

;;;; data is formatted and returned to GQP

```
<1  (GQP  (((ALUMNI  LAST-NAME)  (ALUMNI  FIRST-NAME)  (ALUMNI
DEGREE)
                (COMPANY STATE)  (FINANCE PROFIT)  (FINANCE CURRENCY)
              (FINANCE MULT))
                ("George"  "Ernest"  "SM 1979"  "NJ"  "33598.0"  "$-US"
                "million")
                ("George"  "Ernest"  "SM 1979"  "MA"  "33598.0"  "$-US"
               "million")))
```

In this chapter, we demonstrated the feasibility of MERGE for providing data connectivity for CIS/TK. Unfortunately, due to time constraints and problems in the data reconciliation facilities, we could not show these tools in action. In the next chapter, we present the conclusion of our work, and point towards some possible future work in developing MERGE.

8 CONCLUSION

In this thesis, the design of a distributed database management system for providing data connectivity for CIS/TK was presented. This was motivated by the goal of providing a single, integrated environment to access and combine data from various heterogeneous, pre-existing databases. The key difference between MERGE and other Distributed DBMS is that MERGE is designed with the intent of serving as a foundation for further work in semantic connectivity. In order to achieve this, it was necessary to design MERGE to be extensible, and to define interfaces for the addition of tools for semantic data reconciliation. In this chapter, we first discuss how the design of MERGE faired in satisfying these goals. Then, we present some possible future work for extending MERGE.

8.1 Insights

Several insights about the design of MERGE were gained during the implementation of the system as well as during the development of the PAS application for testing the system.

The separation of the GQP into two parts: the query parser and the query router proved to be a very effective design choice. It provided a very clear way to describe the system -- something which was found to be lacking in the preliminary prototype. This was mainly because the MERGE GQP design corresponded well to the tasks involved in global query processing, that is, planning all the subtasks that need to be done and actually executing these tasks. This separation of the GQP will allow future developers to change the router without affecting the parser, if such a need arises due to particular needs of the application.

The facility for automatically selecting databases for a global query proved to be a big relieve for both casual users and developers of the system. In addition, the use of a changeable set of rules for performing database selection allowed one to change the criteria for determing an access path depending on the application and the requirements of the user. When testing the system with the PAS application, there were several times when we had to change the rules because of the type of data we wished to retrieve. By separating the criterias for database selection from the selection mechanism itself, we can in the future expand upon the current default rules without modifications to the system, something not possible with systems that imbed the rules within the selection mechanism.

On the other hand, creating interfaces for data reconciliation within the GQP proved to be a harder issue than at first thought, especially when the range of possible tools and their implementations for data reconciliation are unknown. Nevertheless, the two basic ideas about performing data reconciliation before data is retrieved and after data is retrieved proved to be useful guidelines for interfacing to such tools. Difficulties arise when data reconciliation required the coordination of both pre-data retrieval and post-data retrieval enhancements.

Focusing on the other component of MERGE, that is the data model, we found that the distinction between the structural properties and the semantic properties of data allowed us to tackle each problem separately with considerable success. This was because the structural properties remained fairly stable and once a global schema was created, there was rarely any need to modify it. As for the semantic properties, even with the simple PAS application, we were constantly finding more examples of different types of semantic conflicts. This convinced us that the MERGE data model, with its goal of extensibility, was an appropriate representation scheme.

During the implementation of this thesis, we attempted to build some simple data reconciliation tools for resolving synonyms and translations. However, we found that without a domain mapping system, that is, a facility that allows one to express the properties of the underlying data, like integer, string or character, we were really hampered in our attempts to build such tools.

8.2 Future Work

The insights gained point to some possible future work for developing MERGE. Firstly, we think at least some form of domain mapping support is needed to express the basic properties of the data, perhaps like integers and string identification. Other areas include the development of a wider range of selection rules to optimize access time or costs. Also, the development of a query language that can provide more operations would be useful, for example an extended version of the SQL language.

Other possible areas of future work that are not directly within MERGE but relevant, include the development of data reconciliation tools, with which we can test the GQP for its data reconciliation interfacing abilities.

Work in this thesis on providing data connectivity for CIS/TK and serving as a foundation for semantic connectivity has only scratched the surface of many interesting issues. Nevertheless, we feel that the contribution of this thesis will allow researchers to explore the intriguing problems in semantic connectivity without having to be burdened with the tasks of getting the data from various dissimilar machines.

REFERENCES

[BHA 87] Bhalla S., Prasad B., Gupta A., and Madnick S., "A Technical Comparison of Distributed Heterogeneous Database Management Systems," 1987.

[BRO 84] Brodie, M.L., "On the Development of Data Models," *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, 1984.

[CHA 88] Champlin, A., "Interfacing Mutiple Remote Databases in an Object-Oriented Framework", *Bachelor's Thesis*, MIT, May 1988.

[CHE 76] Chen, P. "The Entity-Relationship model: Towards a unified view of data," *ACM Trans. Database Syst. 1*, 1 March, 1976.

[DAT 87] Date, C.J., *The SQL Standard*, 1987.

[DEE 82] Deen, S.M. "Distributed Databases - An Introduction," *Distributed Data Bases*, 1982.

[GAN 89] Gan, F. "An Architecture Design and Implementation of a Communication Server for Disparate Databases." *B.S.Thesis*, MIT, 1989.

[GER 89] Gerber, H. "Optimizing Information Retrieval For Disparate Menu Driven Database Systems." *B.S. Thesis*, MIT, 1989.

[HOR 88] Horton, D.C., Madnick, S.E., Wang, Y.R., "Inter-Database Instance Identification in Composite Information Systems, "*Proceedings of the Twenty-Second Annual Hawaii International Conference on Systems Sciences*, January, 1989.

[HOR 88-2] Horton, D.C., "An Object-Oriented Approach Towards Enhancing Logical Connectivity in a Distributed Database Environment," *M.S. Thesis*, MIT Sloan School, 1988.

[KIN 84] King, R., McLeod D., "A Unified Model and Methodology for Conceptual Database Design" *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, 1984.

[LEV 87] Levine S., "Interfacing Objects and Databases", *M.S. Thesis*, MIT, 1987.

[LIN 87] Lindsay B., "A Retrospective of R*": A Distributed Database Management System," *Proceedings of the IEEE*, Vol 75 , No5, May 1987.

[MAD 88-1] Madnick, S.E., Wang, Y.R., "A Framework of Composite Information Systems for Strategic Advantage," *Proceedings of the Twenty-First Annual Hawaii International Conference on Systems Sciences*, January 1988.

[MAD 88-2] Madnick, S.E., Wang Y.T., "Evolution Towards Strategic Applications of Databases Through Composite Information Systems," *Connectivity Among Information Systems*, Vol 1, MIT, Cambridge MA, 1988.

[MCC 88] McCay, B.C., "Translation Facility of the Composite Information System Tool Kit, Version 1.0," *Technical Report CIS-88-10*, MIT, Aug. 1988.

[NEU 82] Neuhold, E. J., Walter, B., "An Overview of the Architecture of the Distributed Data Base System "POREL"", *Distributed Data Bases*, September 1982.

[PAG 89] Paget, M.L., "A Knowledge-Based Approach toward Integrating International On-line Databases", *M.S. Thesis*, MIT, 1989.

[ROS 82] Rosenberg, R.L, Landers, T., "An Overview of Multibase", *Distributed Databases*, September 1982.

[SCH 82] Schneider, H.J., *Distributed Data Bases*, September 1982.

[SHA 84] Shaw, M., "The Impact of Modelling and Abstraction Concerns on Moderm Programming Languages" *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, 1984.

[STO 84] Stonebraker, M., "Adding Semantic Knowledge to a Relational Database System," *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, 1984.

[WAN 88-1] Wang, Y.T., Madnick, S.E., "Logical Connectivity: Applications, Requirements, and An Architecture," MIT, 1988.

[WAN 88-2] Wang, Y.T., Madnick, S.E., Horton D.C., and Wong, T.K. "Concept Agents in CIS/TK: A Tool Kit for Composite Information Systems," *Proceedings of the International Computer Symposium*, Taiwan, December 1988.

[WEG 86] Weger, P., "Perspectives on Object-Oriented Programming," *Technical Report No. CS-86-25*, Brown University, December 1986.

[WON 88] Wong, T.K., Alford, M. "The CIS/TK Implementation V1.0," *Technical Report CIS-88-11*, MIT, August 1988.

APPENDIX A.1 - Schema Definition for PAS

```
;;;; MIT PLACEMENT OFFICE GLOBAL SCHEMA
;;; This file tests the schema definition language
;;; 3 entities: alumni, company and finance
;;; 2 relationships: works_for, finance_info


(create-schema mit-placement)

(create-entity alumni
             :attributes ((first-name (local2e alumnitb first_name))
                          (last-name (local2e alumnitb last_name))
                          (degree (local2e alumnitb degree))
                          (company (local2e alumnitb company_name)))
             )

(create-entity company
             :attributes ((name (oracle2e companytbl company_name))
                          (position (oracle2e companytbl position))
                          (date (oracle2e companytbl visit_day))
                          (sic (oracle2e companytbl sic_code)
                             (local2e siccodetb sic_code))
                          (industry (local2e siccodetb industry)))
             :table-relations ((merge (oracle2e companytbl)
                                (local2e siccodetb)
                                on (= (oracle2e companytbl sic_code)
                                      (local2e siccodetb sic_code))))
             )

(create-entity finance
             :attributes ((company (disclosure2e geninfo co)
                               (dataline2e data company_name))
                          (code (dataline2e data code))
                          (compno (disclosure2e geninfo compno)
                              (disclosure2e gennum compno))
                          (revenue (disclosure2e gennum ni)
                               (dataline2e data efo))
                          (profit (disclosure2e gennum ns)
                              (dataline2e data sales))
                          (currency (disclosure2e geninfo curr)
                                (dataline2e data currency))
                          (mult (disclosure2e gennum mult))
                          (period (disclosure2e gennum cf)
                              (dataline2e data periodending)))
             :table-relations ((merge (disclosure2e geninfo)
                                (disclosure2e gennum)
                                on (= (disclosure2e geninfo compno)
                                      (disclosure2e gennum compno))))
             )


(create-relation works_for
             :entity-from alumni
             :entity-to company
             :join (= (alumni company)
                    (company name)))
```

```
(create-relation finance_info
             :entity-from company
             :entity-to finance
             :join (= (company name)
                    (finance company)))
```

APPENDIX B.1 - GLOBAL RETRIEVAL LANGUAGE

The following is a BNF of the GRL syntax. Letters in caps are the actual syntax. Letters in italic refer to the left-hand side of the BNF equations. The parenthesis "(" and ")" are part of the syntax of GRL. Letters enclosed between "{ ... }" are optional. Letters enclosed between "[.. | .. | ..]" means that one of several options can be used which are divided by "|".

join-query
```
        ::=     (JOIN select-query join-query
                    {ON label })
```
select-query
```
        ::=     (SELECT entity [(att... att) | *]
                    {WHERE select-condition})
```

label
```
        ::=     name of relation object
```

entity
```
        ::=     entity object
```

att
```
        ::=     attribute
```

select-condition
```
        ::=     (binary-op select-condition select-condition ) |
                ( = (entity att ) val )
```

binary-op
```
        ::=     AND | OR
```

val
```
        ::=     value surrounded by double quotes
```

APPENDIX B.2 -SCHEMA DEFINITION LANGUAGE

create-schema

```
        ::=     (CREATE-SCHEMA schema)
```

create-entity

```
        ::=     (CREATE-ENTITY entity
                        :ATTRIBUTES (map ... map)
                        {:TABLE-RELATIONS (tb-rel ... tb-rel)})
```

create-relation

```
        ::=     (CREATE-RELATION relation
                        :ENTITY-FROM entity
                        :ENTITY-TO entity
                        :JOIN join-rel )
```

schema

```
        ::=     name of global schema
```

entity

```
        ::=     entity object
```

map

```
        ::=     (att [(lqp tb col) .. (lqp tb col)])
```

lqp

```
        ::=     local query processor object
```

tb

```
        ::=     table
```

```
col
        ::=    column


tb-rel
        ::=    (MERGE source source
                      ON merge-cond) |
               (CONCATENATE source source)


source
        ::=    (lqp tb)


merge-cond
        ::=    (AND  (= (lqp tb col) (lqp tb col))
                      merge-cond) |
               (= (lqp tb col) (lqp tb col))


join-rel
        ::=    (AND  (= (entity  att)  (entity  att))
                      join-rel) |
               (= (entity  att)  (entity  att)
```

APPENDIX C.1 GQP V3.0 CODE

```
;;;; This file implements the Global Query Processor
;;; TK
;;; April 27, 89

;;; declare all the global attributes
(defvar *global-atts*)

;;; gqp
(defun gqp (query)
 (let ((parsed)
          (gs->loc)
          (loc->gs))
   (multiple-value-setq (parsed_query gs->loc loc->gs) (parser query))
   (router parsed_query gs->loc loc->gs)))
```

PARSER MODULES

```
;;;; This file implements the Parser
;;; April 1, 89

;;;; PARSER
;;;
(defun parser (query)
 (let ((p_query)
          (ent_atts)
          (locs)
          (gs->loc)
          (loc->gs))
   ;; expand the query
   (setf p_query (expand_query query))
    ;; get all the attributes in the query
   (setf ent_atts (list_query_atts p_query))
    ;; get all the locs
   (setf locs (get_att_locs ent_atts))
    ;; make the map tables, a cache of all
    ;; attributes and its location used in
    ;; the query
   (setf gs->loc (make_gs->loc locs))
   (setf loc->gs (make_loc->gs (reverse_map locs)))
    ;; make the intermediate query
   (setf p_query (determine_source p_query gs->loc loc->gs))
    ;; enhance the query
   (setf p_query (enhance_query p_query))
   (values p_query gs->loc loc->gs)))


;;;; AUXILIARY FUNCTIONS
;;; list_query_atts
;;; list all the attributes in a expanded query
;;; ((ent att) ... (ent att))
(defun list_query_atts (query)
 (cond ((null query)
           nil)
          ((select? query)
           (get_select_atts query))
          ((join? query)
           (let ((join1 (get_join1 query))
                 (join2 (get_join2 query)))
             (append (list_query_atts join1)
                         (list_query_atts join2))))))
```

QUERY EXPANSION MODULE

```
;;; This file implements the query expansion module
;;; Expansions include:
;;; 1.  expand relation labels
;;; 2.  expand *-option with attributes
;;; 3.  transform query attributes and conditions
;;;     into long-form.
;;; 4.  insert join key attributes


;;; expand_query
;;; expands the query

(defun expand_query (query)
  (let ((exp_query))
    ;; reset global atts
    (reset_global_atts)
    ;; expand query
    (setf exp_query (expand query))

    ;; insert join attributes
    (insert_joinkeys exp_query)))


;;; expand
;;; expands stages 1 to 3 (shown at beginning)

(defun expand (rest_query)
  (cond ((null rest_query)
         nil)
        ((join? rest_query)
         (let ((join1 (get_join1 rest_query))
               (join2 (get_join2 rest_query))
               (join_on (get_join_on rest_query)))

           ;; expand and make a join statement
           (make_join (expand join1)
                      (expand join2)
                      (expand_relation join_on
                                       (get_select_entity join1)
                                       (get_select_entity
                                         (get_first_join join2))))))
        ((select? rest_query)
         (let* ((entity (get_select_entity rest_query))
                (atts (get_select_atts rest_query))
                (conds (get_select_conds rest_query))
                ;; combine the atts from condition
                (cond_atts (list_cond_atts conds)))

           ;; expand and make a select statement
           (make_select entity
                        (expand_attributes entity atts cond_atts)
                        (expand_conds entity conds))))))
```

```
;;; ==================
;;; Expand Attributes
;;; ==================

;;; expand_attributes
;;; expand the attributes of a select statement

(defun expand_attributes (entity atts cond_atts)
  (cond ((*-option? atts)
          (let* ((*-atts (get_*_atts entity))
                  (ent_atts (add_entity entity *-atts)))
            ;;; saves original atts for later formatting
            (save_global_atts ent_atts)
            ent_atts))
         (t
          (let ((ent_atts (add_entity entity atts)))
            ;;; saves original atts for later formatting
            (save_global_atts ent_atts)
            (add_entity entity (union-equal cond_atts atts))))))


;;; get_*_atts
;;; gets all the attributes for that entity

(defun get_*_atts (entity)
  (get_entity_attributes entity))


;;; ==================
;;; Expand Conditions
;;; ==================

;;; expand_conds
;;; expands the conditions

(defun expand_conds (entity conds)
  (cond ((null conds) nil)
         ((simple_cond? conds)
          (expand_simple_cond entity conds))
         ((nested_cond? conds)
          (let ((op (first conds))
                (cond1 (second conds))
                (cond2 (third conds)))
            (make_nested_cond op
                              (expand_conds entity cond1)
                              (expand_conds entity cond2))))))


;;; expand_simple_cond
;;; expand simple condition

(defun expand_simple_cond (entity cond)
  (let ((op (first cond))
         (att (second cond))
         (val (third cond)))
    (make_simple_condition op
                    (first (add_entity entity (list att)))
                    val)))
;;; ============
;;; Expand Relation
;;; ============

;;; expand_relation
;;; expands a relation label into the actual join information if not
;;; already specified.  If no label is specified, a relation is chosen.

(defun expand_relation (relation entity1 entity2)

 ;; relation not specified
 (cond ((null relation)
        (get_relation_join (choose_relation entity1 entity2)))
       ;; actual relation join information
       ((relation_join? relation)
        relation)
       ;; otherwise it is a relation label
       (t
        (get_relation_join relation))))


;;; choose_relation
;;; choose the relation if not specified in the query
;;; the first relation that matches in both the entities is chosen.
(defun choose_relation (entity_to entity_from)
 (let ((rels_1 (get_entity_relations entity_to))
       (rels_2 (get_entity_relations entity_from))
       (same_rels))

   ;; find the shared relations between entities
   (setf same_rels (intersection-equal rels_1 rels_2))

   ;; choose the first one on the list
   (first same_rels)))


;;; ==================
;;; Insert Join Attributes
;;; ==================

;;; insert_joinkeys
;;; takes an expanded query and inserts the join attributes not
;;; already found in the query

(defun insert_joinkeys (query)
 (cond ((null query) nil)
       ((select? query)
        query)
       ((join? query)
        (let* ((join_on (get_join_on query))
               (join1 (insert_join1_atts (get_join1 query)
                                         join_on))
               (join2 (insert_join2_atts (get_join2 query)
                                         join_on)))
          (make_join join1
                     join2
```

```
                         join_on)))))


;;; insert_join1_atts
;;; insert the join attributes for the first part of the join
;;; statement.

(defun insert_join1_atts (query join_on)
  (let* ((atts (list_atts join_on))
          (entity (get_select_entity query))
          (select_atts (select_atts entity atts)))
   (insert_select_atts select_atts query)))


;;; insert_join2_atts
;;; insert the join attributes for the second part of a join
;;; statement.

(defun insert_join2_atts (query join_on)
 (cond ((select? query)
          (let* ((join_atts (list_atts join_on))
                   (ent (get_select_entity query))
                   (select_atts (select_atts ent join_atts)))
           (insert_select_atts select_atts query)))
         ((join? query)
          (let* ((join1 (get_join1 query))
                   (join2_on (get_join_on query)))
           (make_join (insert_join2_atts join1 join_on)
                        (insert_join2_atts (get_join2 query) join2_on)
                        join2_on)))))



;;; insert_select_atts
;;; insert attributes into a select statement

(defun insert_select_atts (atts query)
 (let* ((query_atts (get_select_atts query))
          (all_atts (union-equal atts query_atts)))
   (make_select (get_select_entity query)
                        all_atts
                        (get_select_conds query))))


;;;=====================
;;; auxiliary functions
;;;=====================

;;; add_entity
;;; adds the entity to each item in the list, ie (entity item)

(defun add_entity (entity items)
 (let ((expand_items))
   (dolist (item1 items)
            (setf expand_items (append expand_items
                                          (list (list entity item1)))))
   expand_items))
```

```
;;; get_first_join
;;; gets the first select statement from the second part of a join
;;; statement.

(defun get_first_join (join2)
 (cond ((select? join2)
          join2)
         ((join? join2)
          (get_join1 join2))))


;;; list_atts
;;; list all the attributes in the on part of a join statement.
(defun list_atts (join_on)
 (let ((atts))
  (labels ((list_atts1 (rest_join_on)
                       (cond ((null rest_join_on))
                             ((simple_cond? rest_join_on)
                               (setf atts
                                     (append atts
                                             (rest rest_join_on))))
                             ((nested_cond? rest_join_on)
                               (setf atts
                                     (append atts
                                             (list (second
                                                    (second
                                                     rest_join_on)))))
                               (list_atts1 (third rest_join_on))))))
          (list_atts1 join_on))))

;;; list_cond_atts
;;; returns all the attributes in a condition list
(defun list_cond_atts (cond)
 (cond ((null cond) nil)
         ((simple_cond? cond)
          (list (second cond)))
         ((nested_cond? cond)
          (let ((cond1 (second cond))
                (cond2 (third cond)))
            (union-equal (list_cond_atts cond1)
                         (list_cond_atts cond2))))))


;;; select_atts
;;; returns all attributes with the same entity as specified

(defun select_atts (entity atts)
 (let ((selected_atts))
  (dolist (att atts)
          (if (equal entity (first att))
               (setf selected_atts (union-equal selected_atts
                                                (list att)))))
  selected_atts))
```

```
;;;; AUXILIARY FUNCTIONS

;;; save_global_atts
;;; sets the attributes defined by the user into *global-atts*
;;; to be used for formatting later, ie before the data is returned
;;; to the user
(defun save_global_atts1 (query)
  (setf *global-atts* (get_global_atts query)))

(defun save_global_atts (atts)
  (setf *global-atts* (append *global-atts* atts)))

(defun reset_global_atts ()
  (setf *global-atts* nil))

;;; get_global_atts
;;; returns a list of all the global attributes
(defun get_global_atts1 (query)
  (cond ((null query)
          nil)
         ((join? query)
          (let* ((join1 (get_join1 query))
                  (join2 (get_join2 query)))
            (append (get_global_atts join1)
                    (get_global_atts join2))))
         ((select? query)
          (let ((atts (get_select_atts query)))
            atts))))
```

CREATE ACCESS PLAN MODULES

```
;;; This implements the create access plan module
;;; this module compiles a global query in the following stages:
;;; 1.  find all the possible access paths
;;; 2.  select a path/paths based on a set of selection rules
;;; 3.  compile the query into an intermediate query language


;;; determine_source
;;; determines the access path for a global query in an intermediate query
;;; language.

(defun determine_source (query gs->loc loc->gs)
 (cond ((null query)
         nil)
        ((join? query)
         (list 'join
             (determine_source (get_join1 query) gs->loc loc->gs)
             (determine_source (get_join2 query) gs->loc loc->gs)
              'on
             (get_join_on query)))
        ((select? query)
         (let ((atts (get_select_atts query))
             (conds (get_select_conds query))
              (joins (get_entity_table_rels
                       (get_select_entity query)))
              (access_paths)
             (selected_paths))

          ;; find all the possible access paths
          (setf source_list
                 (find_sources atts joins gs->loc loc->gs))

          ;; select the access paths
          (setf access_paths
                 (select_source source_list atts joins gs->loc loc->gs))

          ;; compile the global query
          (make_intquery atts conds joins access_paths gs->loc loc->gs)))))
```

FIND SOURCE MODULE

```
;;;; This file implements the find sources module
;;; February 23, 89

;;; find_sources
;;; description: returns all the possible sources where all the
;;; attributes can be found.

(defun find_sources (atts joins gs->loc loc->gs)

  ;;; get all possible sources
  (let* ((all_sources (list_combs (list_sources atts gs->loc)
                                  (list_joins joins))))

    ;;; filter sources that cannot satisfy all the attributes
    (filter_combs all_sources atts gs->loc loc->gs)))



;;; list_combs
;;; description: list all possible combinations. if the join
;;; information is nil or does not correspond to the elts, then only
;;; the single combinations are listed.  if the elt list is nil, then
;;; nil is returned.
;;; accepts: (elt ... elt) and ((elt1 elt2) .. (elt1 elt3))
;;; returns: ((elt1) (elt2) (elt3) (elt1 elt2) ... (elt1 elt3))

(defun list_combs (elts join)
 (let ((all-list))
   (labels ((list-elt (elt1 rest-elts)
                      (let ((elist))
                          (dolist (rest-elt rest-elts)
                                  (cond ((elt_join? (first (last elt1))
                                                     rest-elt join)
                                         (setf elist
                                              (append elist
                                                      (list
                                                      (append
                                                          elt1 (list
                                                             rest-elt)))))))))
                          elist))

            (list-elt2 (combs)
                         (let ((new-combs))
                          (cond ((= (length (first combs))
                                         (length elts)))
                                (t
                                 (dolist (c combs)
                                           (setf new-combs
                                               (append new-combs
                                                       (list-elt
                                                           c
                                                           (list_after
                                                            (first (last c))
                                                            elts)))))
                                 (if new-combs
                                     (progn
                                        (setf all-list (append all-list
                                                               new-combs))
                                        (list-elt2 new-combs)))))))
            (list-elt2 '())
            all-list)))


;;; list_after
;;; description: returns the list of elts after the given elt, if elt
;;; is nil then return the whole list
;;; accepts: elt and (elt1 elt elt3 .. eltn)
;;; returns: list of elts after elt, ie. (elt2 .. eltn)

(defun list_after (elt e-list)
  (cond ((null elt)
          e-list)
        ((eq elt (first e-list))
          (rest e-list))
        (t
          (list_after elt (rest e-list)))))


;;; elt_join?
;;; description: checks whether there is a join between 2 elts
;;; accepts: elt_from elt_to ((elt1 elt2) (elt3 elt4) ...)
;;; returns: t if there is a join

(defun elt_join? (elt_from elt_to joins)
  (let* ((pair (list elt_from elt_to))
          (rpair (reverse pair)))
    (if (or (not elt_from)
            (not elt_to))
        t
      (dolist (j joins)
             (if (or (equal pair j)
                     (equal rpair j))
                 (return t))))))



;;; filter_combs
;;; description: takes all the combinations and filters out those that
;;; do not satisfy the query. specifically, it removes all
;;; possibilities that do not contain all the attributes.

(defun filter_combs (s_combs atts gs->loc loc->gs)
  (let ((f_combs)
        (att_list))

    ;;; for each source combination
    (dolist (sources s_combs)

            ;;; for each source
            (dolist (s sources)

                    ;;; get source attributes
                    (let ((s_atts (get_ent_att
                                    (lookup-3map loc->gs
                                                            (get_lqp s)
                                                            (get_tb s)))))

 ;;; if any attributes in att are found in s_atts then place in
 ;;; att_list if not already there.
                         (dolist (s_att s_atts)
                                (if (member-equal s_att atts)
                                     (setf att_list
                                            (union-equal (list s_att)
                                                          att_list)))))

          ;;; if all the attributes in att_list match the
          ;;; attributes in att then place source combination in
          ;;; s_list
          (if (not (set-difference-equal atts att_list))
               (progn (setf f_combs (append f_combs (list
                                                          sources)))))
          ;;; reset att_list
          (setf att_list nil))

   ;;; return f_combs
   f_combs))



;;; get_ent_att
;;; description: returns list of ent_att given ((col ent_att ...
;;; ent_att) ... (col ent_att ... ent_att))

(defun get_ent_att (cols)
  (let ((ant_list))
    (dolist (c cols)
            (let ((atts (rest c)))
              (dolist (a atts)
                     (setf ant_list (append ant_list (list a))))))
    ant_list))


;;; get_lqp
;;; description: returns lqp
;;; accepts: (lqp tb)

(defun get_lqp (source)
  (first source))


;;; get_tb
;;; description: returns tb
;;; accepts: (lqp tb)

(defun get_tb (source)
  (second source))


;;; list_sources
;;; description: returns all the sources of attributes

(defun list_sources (locs gs->loc)
 (let ((s_list))
   (dolist (l locs)
             (setf s_list (union-equal s_list (get_sources l gs->loc))))
   s_list))


;;; get_sources
;;; description: gets the source for the ent_att from the map table

(defun get_sources (att gs->loc)
 (let ((locs (lookup-2map gs->loc (get-ent att) (get-att att)))
          (sources))
   (dolist (l locs)
             (setf sources (append sources (list (get_source l)))))
   sources))

;;; get_source
;;; gets a source from a location
(defun get_source (loc)
 (list (get_lqp loc) (get_tb loc)))



;;; list_joins
;;; description: returns all the tables with joins between them.

(defun list_joins (join)
 (let ((j_list))
   (dolist (j join)
             (if (merge? j)
                  (setf j_list (append j_list
                                        (list (make_tb_join (get_jtb1 j)
                                                             (get_jtb2 j)))))))
   j_list))


;;; make_tb_join
;;; description; takes two tables and puts them in a list

(defun make_tb_join (tb1 tb2)
 (list tb1 tb2))

;;; entity-attribute operators
(defun get-att (ent-att)
 (second ent-att))

(defun get-ent (ent-att)
 (first ent-att))
```

SELECT SOURCE MODULE

```
;;; select_source
;;; description: takes a set of possible source combinations and
;;; selects a combination or a concatenation of combinations depending
;;; on the selection rules given.

(defun select_source (sources atts joins gs->loc loc->gs)
 (let ((srules (get_srules))
         (chosen_sources sources))
   (cond ((only_comb? chosen_sources)
            chosen_sources)
           (t
            (dolist (r srules)
                   (let ((r_sources (apply r (list chosen_sources))))
                     (cond ((null r_sources))
                             ((only_comb? r_sources)
                              (setf chosen_sources r_sources)
                              (return))
                             ;;; more than one combination
                             (t
                             (setf chosen_sources (ask_user? r r_sources))
                             (return)))))


            ;;; return the chosen sources
            chosen_sources))))



;;; selection rule operators
;;; only_combs?, same_table?, same_database?, least_tables?, ask_user?

;;; only_combs?
;;; description: checks whether there is only one combination. Returns
;;; the combination if only one, else nil.

(defun only_comb? (sources)
 (if (= (length sources) 1)
     sources
   nil))



;;; same_database?
;;; description: returns all combinations that come from one database.
;;; Returns nil if there are no source combinations that come from one
;;; dataabse.

(defun same_database? (sources)
 (let ((same_dbs))
   (dolist (s sources)
           (let ((db (get_lqp (first s)))
                   (same_flag t))
             (dolist (c s)
                    (if (not (equal (get_lqp c) db))
                          (progn (setf same_flag nil)
                                 (return))))
```

```
               (if same_flag
                    (setf same_dbs (append same_dbs (list s)))
                   (setf same_flag t))))
   same_dbs))


;;; same_table?
;;; description: returns the sources that come from the same source,
;;; ie same (lqp tb). returns nil if none.

(defun same_table? (sources)
 (let ((same_tbs))
   (dolist (s sources)
             (if (only_comb? s)
                   (setf same_tbs (append same_tbs (list s)))))
   same_tbs))


(defun same_table1? (sources)
 (let ((same_tbs))
   (dolist (s sources)
             (let ((tb (first s))
                      (same_flag t))
               (dolist (c s)
                         (if (not (equal c tb))
                                (progn (setf same_flag nil)
                                         (return))))
               (if same_flag
                    (setf same_tbs (append same_tbs (list s)))
                   (setf same_flag t))))
   same_tbs))


;;; least_tables?
;;; description: returns the combination with the least number of
;;; sources.

(defun least_tables? (sources)
 (let ((least_sources)
         (source_no)
         (min_no))


   ;; calculate the length of each combination and make a list in
   ;; source_no
   (setf source_no (mapcar #'length sources))

   ;;; find the min length
   (setf min_no (apply #'min source_no))

   ;;; search through the list for combinations with this min length
   (dolist (s sources)
             (if (= (length s) min_no)
                   (setf least_sources (append least_sources (list s)))))

   ;;; return the combinations with the least number of sources
   least_sources))
```

```
;;; least_same_db?
;;; description: a combinatiion of the same_db? and least_tables?
;;; primitives.  Returns the least tables from combinations that have
;;; sources from the same database

(defun least_same_db? (sources)
  (let ((same_db (same_database? sources)))
    (cond ((null same_db)
              nil)
             ((only_comb? same_db)
              same_db)
             (t
              (least_tables? same_db)))))



  ;;; slugs

(defun form_iquery (sources atts joins gs->loc loc->gs)
  sources)

(defun concatenate_iquery (sources atts joins gs->loc loc->gs)
  sources)


;;;; SELECTION SOURCE RULES

;;; default rules
(setf *default_srules*
  (list 'same_table? 'same_database? 'least_tables?))

;;; initialy set to default rules
(setf *selection_srules* *default_srules*)

;;; get_srules
;;; return current rules
(defun get_srules ()
  *selection_srules*)

;;; set_srules
;;; to update rules
(defun set_srules (rules)
  (setf *selection_srules* rules))

;;;; ASK USER

;;; switch status
(defvar *user_switch* nil)

;;; ask_user?
;;; checks whether the user switch status
;;; if the switch off, return all sources
;;; else ask the user
(defun ask_user? (key sources)
  (cond (*user_switch*
            (user_select_source key sources))
           (t
            sources)))

;;;; MIGRATE THIS
;;;; user_select_source
(defun user_select_source (key sources)
  (list (first sources)))
```

MAKE INTERMEDIATE QUERY MODULE

```
;;;; This file implements the intermediate query language
;;; March 19, 1989

;;; intermediate query
;;; 3/4 Primitives:
;;; (1)  (get-table lqp tb (att .. att)
;;;              (and (= att "val")
;;;                   (= att "val")))
;;; (2)  (merge (get-table ...)
;;;             (get-table ...))
;;; (3)  (concatenate (merge ..)
;;;                   (get-table ..))



;;; ===================
;;; Intermediate Query
;;; ===================


;;; make_intquery
;;; compiles a global query into an intermediate query
;;; assumes query is from one entity

(defun make_intquery (atts conds joins sources gs->loc loc->gs)
 (labels ((int (source)

                    ;; simple get-table query
                    (cond ((single_source? source)
                         (make_table_query atts
                                              (make_table_conds conds
                                                                 (list_conds
                                                                  conds)
                                                                 (first source)
                                                                 gs->loc)
                                              (first source)
                                              gs->loc
                                              loc->gs))

                     ;; merge query
                     ((mult_sources? source)
                      (make_merge_query atts
                                             conds
                                             joins
                                             source
                                             gs->loc
                                             loc->gs))))

                ;; concatenate queries if there are multiple
                ;; source combinations
                (concat (source)
                      (cond ((only_comb? source)
                             (int (first source)))
                            (t
                             (make_concat_query
                                        (int (first source))
                                        (list (concat (rest source)))))))
        (concat sources)))



;;; ================
;;; get-table statement
;;; ================


;;; make_table_query
;;; makes a get-table statement

(defun make_table_query (atts conds source gs->loc loc->gs)
 (let ((table_query (list 'get-table
                              (get_lqp source)
                              (get_tb source))))

   ;;; add attributes
   (setf table_query (append table_query
                               (list (make_table_atts atts gs->loc
                                                       source))))

   ;;; add conditions
   ;;; if there are conditions

   (if conds
       (setf table_query (append table_query
                                   (list 'where conds))))

   ;;; return table_query
   table_query))



;;; make_table_atts
;;; makes the selection of attributes for the get-table

(defun make_table_atts (atts gs->loc source)
 (let ((local_att_list))
   (dolist (a atts)

          ;;; filter the locations that do not come from the source
          (let* ((loc (filter_locs (lookup-2map
                                     gs->loc
                                     (first a)
                                     (second a))
                                    source))
                 (local_att (third loc)))
            (setf local_att_list (append local_att_list
                                          (list local_att)))))
   local_att_list))
```

```
;;; make_table_conds
;;; makes the condition for a get-table statement

(defun make_table_conds (conds source_conds source gs->loc)
 (cond ((null conds) nil)
          ((simple_cond? conds)
           (if (member-equal conds source_conds)
              (convert_cond conds source gs->loc)
             nil))
          (t
           (let ((cond1 (make_table_conds (second conds)
                                                         source_conds
                                                         source
                                                         gs->loc))
               (cond2 (make_table_conds (third conds)
                                                         source_conds
                                                         source
                                                         gs->loc)))
            (if (null cond1)
               cond2
             (if (null cond2)
                   cond1
               (list (first conds) cond1 cond2)))))))



;;; convert_cond
;;; takes a condition and converts it into the equivalent
;;; condition for the local source

(defun convert_cond (cond source gs->loc)
 (let ((ent_att (second cond))
          (loc)
          (new_cond))

   ;; get the actual location
   (setf loc (filter_locs (lookup-2map
                                   gs->loc
                                   (first ent_att)
                                   (second ent_att))
                                  source))

   ;; replace into cond
   (setf new_cond (list (first cond)
                               (third loc)
                               (third cond)))
   new_cond))


;;;================
;;; merge statement
;;;================

;;; make_merge_query
;;; makes a merge statement

(defun make_merge_query (atts conds joins sources gs->loc loc->gs)
(labels ((merge (rest_sources &optional locs)
                (let* ((f_source (first rest_sources))

                       ;; filter the atts not in the source
                       (source_atts (filter_atts
                                     f_source atts gs->loc))

                       ;; create a list of conditions stripped
                       ;; of the nested operators
                       (cond_list (list_conds conds))

                       ;; filter the conds not in the source
                       (source_conds (filter_conds
                                       f_source cond_list gs->loc))

                       ;; create condition for table
                       (table_conds (make_table_conds conds
                                                      source_conds
                                                      f_source
                                                      gs->loc))

                       ;; create the table query
                       (query (make_table_query
                               source_atts
                               table_conds
                               f_source
                               gs->loc
                               loc->gs)))

                 ;;; if there are locs passed from the last pass
                 ;;; add them
                 (if locs
                     (setf query (insert_locs locs query)))

                 (cond ((null (rest rest_sources))
                        query)
                       (t
                        (let* ((on_info (merge_on f_source
                                                  (first
                                                   (rest rest_sources))
                                                  joins))
                               (on_locs (list_locs on_info))
                               (fsource_locs (filter_mlocs
                                              on_locs
                                              f_source))
                               (rsource_locs (set-difference
                                              on_locs
                                              fsource_locs)))

                          ;;; insert locs that are not in the loc list
                          ;;; of the query
                          (setf query (insert_locs fsource_locs query))

                          (list 'merge
                                query
                                (merge (rest rest_sources) rsource_locs)
                                'on
                                on_info))))))
```

```
            (merge sources)))


;;; ================================
;;; concatenate statement
;;; ================================

;;; make_concat_query
;;; makes the concetenate query

(defun make_concat_query (query1 rest_queries)
  (cond ((null rest_queries)
           query1)
          (t
           (list 'concatenate query1
               (make_concat_query (first rest_queries)
                                        (rest rest_queries))))))



;;; ================================
;;; auxiliary functions
;;; ================================

;;; checks whether the source combination is single

(defun single_source? (sources)
  (if (= (length sources) 1)
      t
    nil))


;;; checks whether there are multiple combinations

(defun mult_sources? (sources)
  (if (= (length sources) 1)
      nil
    t))



;;; filter_locs
;;; description: takes a list of source locations for an ent_att and
;;; returns those specified by the source. Only returns one; the last
;;; one on the list that matches

(defun filter_locs (locs source)
  (let ((floc))
    (dolist (l locs)
           (if (and (equal (get_lqp l) (first source))
                      (equal (get_tb l) (second source)))
                 (setf floc (append floc l))))
    floc))
```

```
;;; filter_conds
;;; description: filters all conditions that do not match the source.

(defun filter_conds (source conds gs->loc)
 (let ((f_conds))
  (dolist (c1 conds)
          (let* ((att (second c1))
                 (locs (lookup-2map gs->loc
                                    (first att)
                                    (second att))))
            (if (filter_locs locs source)
                (setf f_conds (append f_conds (list c1))))))
  f_conds))


;;; list_conds
;;; description: strips the "and" and "or" operators from the
;;; conditions and returns only the conditions.

(defun list_conds (conds)
 (cond ((null conds)
        nil)
       ((nested_cond? conds)
        (append (list_conds (second conds))
                (list_conds (third conds))))
       ((simple_cond? conds)
        (list conds))))


;;; list_locs
;;; description: takes the join information and returns a list of
;;; locations stripped of the "and" operator.

(defun list_locs (locs)
 (let ((loc_list))
  (labels ((strip (rest_locs)
                  (cond ((null rest_locs))
                        ((and_join? rest_locs)
                         (setf loc_list
                               (append loc_list
                                       (rest (second rest_locs))))
                        (strip (third rest_locs)))
                        ((simple_join? rest_locs)
                         (setf loc_list
                               (append loc_list
                                       (rest rest_locs)))))))
          (strip locs)
          loc_list)))


;;; insert_locs
;;; description: inserts a set of locations into a query.  If the
;;; location already exist in the query, then do nothing.  All the
;;; locations must come from the same source.  The query must be a
;;; simple table query.

(defun insert_locs (locs query)
(let ((query_cols (fourth query))  ;;; ** substitute with inter q
        (locs_cols (mapcar #'third locs)))
  (append (list (first query)
                 (second query)
                 (third query)
                 (union-equal query_cols locs_cols))
           (cddddr query))))



;;; filter_mlocs
;;; returns a list of locs that has the same source as the argument source.

(defun filter_mlocs (locs source)
  (let ((floc))
    (dolist (l locs)
           (if (and (equal (get_lqp l) (first source))
                      (equal (get_tb l) (second source)))
                  (setf floc (append floc (list l)))))
    floc))


;;; merge_on
;;; description: makes the on part of the merge query. Expects only
;;; one merge information to be found. If nothing found, returns nil.
;;; At some point, it should complain that nothing was found. *** but
;;; not here.

(defun merge_on (source1 source2 joins)
  (let ((source_join)
         (s1_2 (list source1 source2)))
    (dolist (j1 joins)
           (let ((j1_sources (list (second j1)
                                    (third j1))))
             (if (and (or (equal s1_2 j1_sources)
                             (equal (reverse s1_2) j1_sources))
                        (merge? j1))
                   (progn (setf source_join (fifth j1))
                           (return)))))
    source_join))


;;; filter_atts
;;; description: filters the attributes that do not belong to the
;;; location specified by source

(defun filter_atts (source atts gs->loc)
  (let ((f_atts))
    (dolist (att1 atts)
           (let ((locs (lookup-2map gs->loc
                                    (first att1)
                                    (second att1))))
             (if (filter_locs locs source)
                  (setf f_atts (append f_atts
                                        (list att1))))))
    f_atts))
```

QUERY ENHANCER MODULE

```
;;;; This file implements the query enhancer submodule
;;; April 15, 89

;;; enhance_query
(defun enhance_query (query)
 (cond ((null query)
           nil)
          ((get-table? query)
           (enhance_table query))
          ((merge? query)
           (enhance_merge query))
          ((concatenate? query)
           (enhance_concatenate query))
          ((join? query)
           (enhance_join query))))

;;; enhance_table
;;; only synonym handling
(defun enhance_table (query)
 (let ((lqp (get_table_lqp query))
          (tb (get_table_tb query))
          (atts (get_table_atts query))
          (syns))
   (make_syns_query query lqp tb atts)))


;;; enhance_merge
;;; enhanced with translations
(defun enhance_merge (query)
 (let* ((merge_on (get_merge_on query))
           (merge_list (list_mergeon merge_on)))
   (make_mergetrans_query
    (make_merge (enhance_query (get_merge1 query))
                        (enhance_query (get_merge2 query))
                        (get_merge_on query))
     merge_list)))


;;; enhance_concatenate
;;; synonym handling
(defun enhance_concatenate (query)
 (make_concatenate (enhance_query (get_concat1 query))
                              (enhance_query (get_concat2 query))))


;;; enhance_join
;;; translations
(defun enhance_join (query)
 (let ((join_on (get_join_on query)))
   (make_jointrans_query
    (make_join (enhance_query (get_join1 query))
                    (enhance_query (get_join2 query))
                    join_on)
     (list_joinon join_on))))
```

```
;;;; ENHANCING FACILITIES INTERFACE
;;; enhances the query with synonym handling
(defun make_syns_query (query lqp tb atts)
 (cond ((syns_on?)
         (let ((syns))
           (dolist (att atts)
                   (let ((syn_tb (get_syn_table (eval (get-global-syntb))
                                                         lqp tb att)))
                     (if syn_tb
                         (setf syns (append syns
                                                         (list (list att syn_tb)))))))
           (make_syn_query (make_table lqp tb atts)
                           syns)))
         (t
         query)))


;;; make_mergetrans_query
;;; enhance the merge query with translations
(defun make_mergetrans_query (query merge_list)
 (cond ((trans_on?)
         (let ((trans))
           (dolist (m merge_list)
                   (let ((trans_op (get_trans_op (get-global-transtb)
                                                              (first m) (second m))))
                     (if trans_op
                         (setf trans (append trans
                                                         (list (list m
                                                                        trans_op)))))))
           (make_trans_merge query trans)))
         (t
         query)))


;;; make_jointrans_query
;;; enhances the query with translations
(defun make_jointrans_query (query join)
 (cond ((glotrans_on?)
         (let ((trans))
           (dolist (j join)
                   (let ((trans_op (get_trans_op (get-global-transtb)
                                                              (first j) (second j))))
                     (if trans_op
                         (setf trans (append trans
                                                         (list (list j trans_op)))))))
           (make_trans_join query trans)))
         (t
         query)))


;;;; AUXILIARY FUNCTIONS

;;; list_mergeon
;;; list all the pairs of merge on
(defun list_mergeon (merge_on)
 (cond ((merge_and? merge_on)
         (let ((and1 (merge_and1 merge_on))
               (and2 (merge_and2 merge_on)))
```

```
          (append (list (list (second and1) (third and1)))
                  (list_mergeon and2))))
         (t
          (list (list (second merge_on) (third merge_on))))))


;;; list_joinon
;;; list the entity-attribute pairs
(defun list_joinon (join_on)
 (cond ((join_and? join_on)
         (let ((and1 (join_and1 join_on))
               (and2 (join_and2 join_on)))
          (append (list (list (second and1) (third and1)))
                  (list_joinon and2))))
         (t
          (list (list (second join_on) (third join_on))))))
```

ROUTER MODULES

```
;;;; This file implements the router module
;;; April 15,89

;;; router
(defun router (query gs->loc loc->gs)
 (format_data (parser_router query gs->loc loc->gs)))



;;; parser_router
;;; routes a query from the parser
(defun parser_router (query gs->loc loc->gs)
 (cond ((join? query)
         (let ((join1 (get_join1 query))
               (join2 (get_join2 query))
               (join_on (get_join_on query))
               (data1))
          ;; get data for the first part
          (setf data1 (filter_gs_data (query_router join1) loc->gs))
          ;; convert to global schema terms
          (setf data1 (convert_gs_data data1 loc->gs))
          ;; insert data from first part into 2nd part
          (setf join2 (insert_query_jdata data1 join2
                                          (list_joinon join_on)
                                          gs->loc))
          ;; join data
          (join_data data1
                     (parser_router join2 gs->loc loc->gs)
                     join_on)))
        (t
         (convert_gs_data (filter_gs_data
                           (query_router query) loc->gs)
                          loc->gs))))
```

QUERY ROUTER MODULE

```
;;;; This file implements the query router
;;; April 17, 89


;;; query_router
;;; routes an intermediate query
(defun query_router (query)
 (cond ((merge? query)
          (route_merge query))
         ((concatenate? query)
          (route_concatenate query))
         ((get-table? query)
          (route_table query))))


;;; route_table
;;; routes a table query
(defun route_table (query)
 (let ((data)
         (lqp (get_table_lqp query))
         (tb (get_table_tb query))
         (atts (get_table_atts query))
         (conds (get_table_conds query)))
   ;; retrieve data from lqp
   (setf data (get_data lqp tb atts conds))
   ;; convert cols and add it to the data
   (append (list (convert_colnames lqp tb atts))
             (rest data))))


;;; route_merge
;;; routes a merge query
(defun route_merge (query)
 (let ((merge1 (get_merge1 query))
         (merge2 (get_merge2 query))
         (merge_on (get_merge_on query))
         (data1)
         (data2))
   ;; get data for first part
   (setf data1 (route_table merge1))
   ;; insert key data from data1 into merge2
   (setf merge2 (insert_query_data data1 merge2 merge_on))
   ;; get data for second part
   (setf data2 (query_router merge2))
   ;; merge data
   (merge_data data1 data2 merge_on)))


;;; route_concatenate
;;; routes a concatenate query
(defun route_concatenate (query)
 (let ((concat1 (get_concat1 query))
         (concat2 (get_concat2 query)))
   (concat_data (route_table concat1)
                      (query_router concat2))))
```

```
;;;; LQP INTERFACE
;;; get_data
;;; interfaces to the LQP object
(defun get_data (lqp tb atts conds)
  (let ((data (send-message lqp :get-data (list tb atts conds))))
    (cond ((lqp-error? data)
              (format t "LQP returned an error"))
             ((null (rest data))
              (error "No data was found for the query"))
             (t
              data))))


;;; lqp-error?
;;; returns t if lqp-error
(defun lqp-error? (data)
  (if (equal (first data) 'lqp-error)
      t
    nil))


;;;; AUXILIARY FUNCTIONS

;;; convert_colnames
;;; converts the header list returned by an LQP to
;;; have the source too, ie (lqp tb att)
(defun convert_colnames (lqp tb att_list)
  (let ((catts))
    (dolist (att att_list)
              (setf catts (append catts (list (list lqp tb att)))))
    catts))
```

FILTER MODULE

```
;;;; This file implements a filter for data returning from the local lqp's
;;; to the router.
;;; April 30, 89

;;; filter_gs_data
;;; filters the data returned from the lqp
;;; all local attributes not defined in the loc->gs
;;; table are filtered
(defun filter_gs_data (data loc->gs)
 (let ((head_list (first data))
        (data_list (rest data)))
  (filter_gs_datalist (filter_gs_head head_list loc->gs)
                      head_list
                      data_list)))

;;; filter_gs_head
;;; filters those attributes not defined in loc->gs
(defun filter_gs_head (head loc->gs)
 (let ((fhead))
  (dolist (h head)
         (let* ((lqp (first h))
                (tb (second h))
                (att (third h))
                ;; get equivalent global schema term
                (gs (lookup-3map loc->gs lqp tb att)))
          ;; if exist, then add to filter list
          (if gs
              (setf fhead (append fhead
                                  (list h))))))
  fhead))

;;; filter_gs_datalist
;;; filters a set of data on the filter_keys
(defun filter_gs_datalist (filter_keys head data)
 (let ((new_data (list filter_keys)))
  (dolist (d data)
         (let ((row))
          (dolist (key filter_keys)
                 (let ((pos (find_data_position head key)))
                      (if pos
                        (setf row (append row
                                          (list (nth pos d)))))))
          (setf new_data (append new_data (list row)))))
  new_data))
```

COMBINE MODULE

```
;;;; This file implements the combine module
;;; April 18, 89
;;; join_data
;;; joins the data - inner join
(defun join_data (j1 j2 on)
 (let ((j1_header (first j1))
          (j2_header (first j2))
          (j1_data (rest j1))
          (j2_data (rest j2))
          (data))
   ;; cartesian product of data
   (setf data (cartesian j1_data j2_data))
   ;; perform a restriction
   (join_restriction (append (list (append j1_header j2_header))
                                   data) on)))

;;; merge_data
;;; merges the data - inner join
(defun merge_data (m1 m2 on)
 (let ((m1_header (first m1))
          (m2_header (first m2))
          (m1_data (rest m1))
          (m2_data (rest m2))
          (data))
   ;; perform a cartesian product of the data
   (setf data (cartesian m1_data m2_data))
   ;; perform a restriction
   (merge_restriction (append (list (append m1_header m2_header))
                                   data) on)))


;;; concat_data
;;; concatenates the data
;;; returns only the first header, expects c1 and c2 to be
;;; aligned
(defun concat_data (c1 c2)
 (let ((head1 (first c1))
         (rest_data1 (rest c1))
         (head2 (first c2))
         (rest_data2 (rest c2)))
   (append (list head1)
            (append  rest_data1 rest_data2))))


;;; cartesian
;;; performs a cartesian product d1xd2
(defun cartesian (d1 d2)
 (let ((product))
   (dolist (d1_elt d1)
            (dolist (d2_elt d2)
                    (setf product (append product
                                          (list (append d1_elt d2_elt))))))
   product))
```

```
;;;; RESTRICTIONS
;;; merge_restriction
;;; performs a restriction on the data
(defun merge_restriction (data cond)
 (let* ((head (first data))
         (rest_data (rest data))
         (restrict_data (list head)))
   ;; for each data elt
   ;; check restriction condition
   (cond ((null cond)
          data)
         ((merge_and? cond)
          (let* ((and1 (merge_and1 cond))
                 (and2 (merge_and2 cond))
                 (op1 (first and1))
                 (operand1 (second and1))
                 (operand2 (third and1))
                 (pos1 (find_data_position head operand1))
                 (pos2 (find_data_position head operand2)))
           (dolist (d rest_data)
               (let ((data_pos1 (nth pos1 d))
                     (data_pos2 (nth pos2 d)))
                (if (equal data_pos1 data_pos2)
                    (setf restrict_data (union-equal restrict_data
                                                     (list d))))))
           (merge_restriction restrict_data and2)))
         (t
          (let* ((operand1 (second cond))
                 (operand2 (third cond))
                 (pos1 (find_data_position head operand1))
                 (pos2 (find_data_position head operand2)))
           (dolist (d rest_data)
               (let ((data_pos1 (nth pos1 d))
                     (data_pos2 (nth pos2 d)))
                (if (equal data_pos1 data_pos2)
                    (setf restrict_data (union-equal restrict_data
                                                     (list d))))))
           restrict_data)))))


;;; join_restriction
;;; performs a join statement restriction
(defun join_restriction (data cond)
 (let* ((head (first data))
         (rest_data (rest data))
         (restrict_data (list head)))
   ;; for each data elt
   ;; check restriction condition
   (cond ((null cond)
          data)
         ((join_and? cond)
          (let* ((and1 (join_and1 cond))
                 (and2 (join_and2 cond))
                 (op1 (first and1))
                 (operand1 (second and1))
                 (operand2 (third and1))
                 (pos1 (find_data_position head operand1))
                 (pos2 (find_data_position head operand2)))
                (dolist (d rest_data)
                        (let ((data_pos1 (nth pos1 d))
                              (data_pos2 (nth pos2 d)))
                         (if (equal data_pos1 data_pos2)
                              (setf restrict_data (union-equal restrict_data
                                                           (list d))))))
                (join_restriction restrict_data and2)))
               (t
               (let* ((operand1 (second cond))
                        (operand2 (third cond))
                        (pos1 (find_data_position head operand1))
                        (pos2 (find_data_position head operand2)))
                (dolist (d rest_data)
                        (let ((data_pos1 (nth pos1 d))
                              (data_pos2 (nth pos2 d)))
                         (if (equal data_pos1 data_pos2)
                              (setf restrict_data (union-equal restrict_data
                                                           (list d))))))
                restrict_data)))))

;;;; AUXILIARY FUNCTION

;;; find_data_position
;;; returns the position of a datum in a returned data list
;;; starts with 0
(defun find_data_position (head key)
 (let ((pos 0)
        (flag nil))
  (dolist (h head)
          (if (equal h key)
               (progn (setf flag t)
                      (return))
           (incf pos)))
  ;; if key not found
  (if flag
       pos
   (error "The key ~A was not found in the header list:~%~A"
          key head))))
 ;;; inserts data from the first part of a merge statement into query
;;; data is expected to come from one source
(defun insert_query_data (data query merge_on)
 (cond ((merge? query)
            ;; only insert data into first part of merge
            (insert_merge_data data query merge_on))
           ((concatenate? query)
            (make_concatenate
             ;; only insert data into first part of concatenate
             (insert_concatenate_data data (get_concatenate1 query) merge_on)
             (get_concatenate2 query)))
           ((get-table? query)
            (let ((source_from (get_source (first (first data))))
                  (source_to (list (get_table_lqp query)
                                   (get_table_tb query))))
             (insert_table_data data source_from
                                source_to query merge_on)))))

;;; insert_merge_data
(defun insert_merge_data (data query merge_on)
 (let* ((merge1 (get_merge1 query))
           (merge2 (get_merge2 query))
           (on (get_merge_on query))
           (head (first data))
           (source_to (list (get_table_lqp merge1)
                            (get_table_tb merge1)))
           (source_from (get_source (first head))))
   (make_merge
    (insert_table_data data source_from source_to merge1 merge_on)
    merge2
    on)))

;;; insert_concatenate_data
;;; inserts data from the first part of a merge into concatenate query
(defun insert_concatenate_data (data query merge_on)
 (let* ((concat1 (get_concatenate1 query))
           (concat2 (get_concatenate2 query))
           (lqp (get_table_lqp concat1))
           (tb (get_table_tb concat1))
           (source_to (list lqp tb))
           (head (first data))
           (source_from (get_source (first head))))
   (make_concatenate (insert_table_data data source_from
                                        source_to concat1 merge_on))))


;;; insert_table_data
;;; inserts data into a table query
;;; expects a get-table query
(defun insert_table_data (data source_from source_to query merge_on)
 (let ((conds (get_table_conds query))
          (lqp (get_table_lqp query))
          (tb (get_table_tb query))
          (atts (get_table_atts query)))
   (make_table lqp tb atts
               (insert_cond_data data source_from
                                 source_to conds merge_on))))
```

```
;;; each elt in a row of data is combined with an "and"
;;; rows are combined with an "or"
;;; the new conditions are combined with the old on "and"
;;; changed "and" to "or" -what's the implications!!!
(defun insert_cond_data (data source_from source_to conds merge_on)
 (let* ((merge_list (list_mergeon merge_on))
         (new_conds (get_conds_data data source_from source_to merge_list)))
   (if (null conds)
         (make_cond_data new_conds)
    (make_table_and  conds (make_cond_data new_conds)))))

;;; get_conds_data
;;; converts the data into a list with the new source
(defun get_conds_data (data source_from source_to merge_list)
 (let* ((key (first data))
         (data_list (rest data))
         (filter_keys (create_key_map key merge_list)))
   ;; filter data not in filter_keys
   (filter_data filter_keys data)))



;;; filter_data
;;; filters unwanted data not in the map_list
;;; also filters out repeated data
(defun filter_data (map_list data)
 (let ((fdata)
         (hdata)
         (data_keys (first data))
         (rest_data (rest data)))
   (dolist (d rest_data)
             (let ((fdata1))
               (dolist (map map_list)
                         (let* ((key (first map))
                                 (pos (find_data_position data_keys key)))
                              (setf fdata1
                                  (append fdata1
                                           (list (nth pos d))))))
              (setf fdata (union-equal fdata  (list fdata1)))))
   (dolist (map map_list)
             (let ((map_key (second map)))
              (setf hdata (append hdata (list map_key)))))
   (append (list hdata) fdata)))


;;; create_key_map
(defun create_key_map (keys map_list)
 (let ((key_map))
   (dolist (key keys)
             (dolist (m map_list)
                        (if (member-equal key m)
                              (setf key_map
                                  (append key_map
                                           (list
                                            (list key
                                                  (first (set-difference m
                                                                   (list key)
                                                                   :test
```

```
                                                                                     #'equal)))))))))
   key_map))


;;; make_cond_data
;;; takes a list ((key1 key2 ..) (data1 data2 ..) (data1 data2 ..) ..)
;;; and makes a condition statement with "or" between rows and "and"
;;; within each row.
(defun make_cond_data (data)
 (let ((keys (first data))
          (data_rows (rest data)))
   (make_cond_rows keys data_rows)))


;;; make_cond_rows
;;; combines each row with "or"
(defun make_cond_rows (key rows)
 (cond ((null (rest rows))
          (make_cond_row key (first rows)))
         (t
          (make_table_or (make_cond_row  key (first rows))
                         (make_cond_rows key (rest rows))))))

;;; make_cond_row
;;; combines elts in a row with "and"
(defun make_cond_row (key row)
 (cond ((null (rest row))
          (list '= (get_loc_col (first key)) (first row)))
         (t
          (make_table_and (list '= (get_loc_col (first key)) (first row))
                              (make_cond_row (rest key) (rest row))))))


;;;; AUXILIARY FUNCTION

;;;; LOCATION ABSTRACTION
(defun get_loc_col (location)
 (third location))
```

CONVERT MODULE

```
;;;; This file implements the convert module
;;; converts a data header into required form at gqp level
;;; April 19, 89

;;; convert_colnames
;;; attaches to each col name the lqp and tb info
(defun convert_colnames (lqp tb cols)
 (let ((clist))
   (dolist (col cols)
             (setf clist (append clist (list (list lqp tb col)))))
   clist))


;;; convert_gs_data
;;; changes the header of the data into global terms
;;; and returns the list of header and data
(defun convert_gs_data (hdata loc->gs)
 (let ((head (first hdata))
           (data (rest hdata)))
   (append (list (convert_gs_atts head loc->gs))
             data)))


;;; convert_gs_atts
;;; converts to global schema attributes
(defun convert_gs_atts (header loc->gs)
 (let ((c_header))
   (dolist (att header)
             (let* ((lqp (first att))
                      (tb (second att))
                      (col (third att))
                      ;; get global schema attribute from
                      ;; map table
                      (gs_att  (lookup-3map loc->gs lqp tb col)))
              (setf c_header (append c_header gs_att))))
   c_header))
```

```
;;;; This file implements the insertion of join data
;;; April 26, 89


;;; insert_query_jdata
;;; inserts join data into second part of the query
(defun insert_query_jdata (data query on_list gs->loc)
 (cond ((join? query)
          (insert_join_jdata data query on_list gs->loc))
         ((merge? query)
          (insert_merge_jdata data query on_list gs->loc))
         ((concatenate? query)
          (insert_concatenate_jdata data query on_list gs->loc))
         ((get-table? query)
          (insert_table_jdata data query on_list gs->loc))))


;;; insert_join
;;; inserts into a join query data from a first part of a join
(defun insert_join_jdata (data query on_list gs->loc)
 (let* ((join1 (get_join1 query))
          (join2 (get_join2 query))
          (on (get_join_on query))
          (head_list (first data))
          (entity_from (get-ent (first head_list))))
   (cond ((get-table? join1)
           (make_join (insert_table_jdata data join1 on_list gs->loc)
                      join2 on))
          ((merge? join1)
           (make_join (insert_merge_jdata data join1 on_list gs->loc)
                      join2 on))
          ((concatenate? join1)
           (make_join (insert_concatenate_jdata data join1 on_list gs->loc)
                      join2 on)))))


;;; insert_table_jdata
;;; inserts join data into a get-table query
(defun insert_table_jdata (data query on_list gs->loc)
 (let ((lqp (get_table_lqp query))
        (tb (get_table_tb query))
        (atts (get_table_atts query))
        (conds (get_table_conds query)))
   (make_table lqp tb atts
                 ;; insert conditions created by the previous
                 ;; join data
                 (insert_cond_jdata data (list lqp tb) conds
                                    on_list gs->loc))))

;;; insert_merge_jdata
;;; inserts data into merge query
(defun insert_merge_jdata (data query on_list gs->loc)
 (let ((merge1 (get_merge1 query))
        (merge2 (get_merge2 query))
        (on (get_merge_on query)))
   ;; insert data into first part of merge
   ;; which is a get-table query
   (make_merge (insert_table_jdata data merge1 on_list gs->loc)
                    merge2
                    on)))


;;; insert_concatenate_jdata
;;; insert joins data into concatenate query
(defun insert_concatenate_jdata (data query on_list gs->loc)
 (let ((concat1 (get_concatenate1 query))
        (concat2 (get_concatenate2 query)))
  (make_concatenate (insert_table_jdata data concat1 on_list gs->loc)
                      concat2)))


;;;; AUXILIARY FUNCTIONS
;;; insert_cond_jdata
;;; inserts join data into condition statement
(defun insert_cond_jdata (data source_to conds on_list gs->loc)
 (let ((new_conds (convert_jdata (get_conds_jdata data
                                                    on_list)
                                 source_to gs->loc)))
  (if (null conds)
        (make_cond_data new_conds)
   (make_table_and conds (make_cond_data new_conds)))))

;;; get_conds_jdata
;;; gets the conditions for data
(defun get_conds_jdata (data on_list)
 (let* ((key_list (first data))
         (data_list (rest data))
         (filter_keys (create_key_map key_list on_list)))
  (filter_data filter_keys data)))

;;; convert_jdata
;;; converts the header list of join data into local attributes
(defun convert_jdata (jdata source_to gs->loc)
 (let ((head (first jdata))
        (data (rest jdata))
        (new_head))
  (dolist (h head)
          (let ((entity (get-ent h))
                 (att (get-att h)))
           (setf new_head
                 (append new_head
                         ;; filter the locs
                         ;; since each global attribute
                         ;; can be mapped to several locs
                         (list
                          (filter_locations
                           (lookup-2map gs->loc entity att) source_to))))))
  (append (list new_head) data)))

;;; filter_locs
;;; filter the locations until only one location
(defun filter_locations (locs source_to)
 (let* ((floc (first locs))
         (source (get_source floc)))
  (if (equal source_to source)
        floc
```

```
    (filter_locs (rest locs) source_to))))



;;; make_cond_data
;;; takes a list ((key1 key2 ..) (data1 data2 ..) (data1 data2 ..) ..)
;;; and makes a condition statement with "or" between rows and "and"
;;; within each row.
(defun make_cond_data (data)
 (let ((keys (first data))
         (data_rows (rest data)))
   (make_cond_rows keys data_rows)))


;;; make_cond_rows
;;; combines each row with "or"
(defun make_cond_rows (key rows)
 (cond ((null (rest rows))
          (make_cond_row key (first rows)))
         (t
          (make_table_or (make_cond_row  key (first rows))
                    (make_cond_rows key (rest rows))))))

;;; make_cond_row
;;; combines elts in a row with "and"
(defun make_jcond_row (key row)
 (cond ((null (rest row))
          (list '= (get (first key)) (first row)))
         (t
          (make_table_and (list '= (get_loc_col (first key)) (first row))
                         (make_cond_row (rest key) (rest row))))))
```

FORMAT MODULE

```
;;;; This file implements the format module
;;; strips off data that was not requested for
;;; April 19, 89

;;; format_data
;;; strips off data from hdata not requested in glob_atts
(defun format_data (hdata)
 (let* ((head (first hdata))
           (data (rest hdata))
           ;; get global atts
           (atts *global-atts*)
           (fdata (list atts)))
  (dolist (d data)
            (setf fdata (append fdata
                                          (list (filter_global_data atts head d)))))
  fdata))

;;; filter_global_data
;;; returns list of data defined by atts
(defun filter_global_data (atts head data)
 (let ((fdata))
  (dolist (att atts)
            (let ((pos (find_data_position head att)))
             ;; expects to find something
             (if pos
                  (setf fdata (append fdata (list (nth pos data)))))))
  fdata))
```

MAPPING (CACHE) MODULES

```
;;;; This file implements 2 map tables to store the mappings from
;;;; global to local schema names, and vice-versa.

;;;; MAKE_GS->LOC
;;; makes the map from global to local terms from list of attributes.
;;; eg. ((alumni name) ... (company name))
(defun make_gs->loc (att_locs)
 (let ((map (make-2keytable)))
   (insert-2map map att_locs)
   map))

;;; make_loc->gs
;;; make a map from local to global terms from atts
(defun make_loc->gs (loc_atts)
 (let ((map (make-3keytable)))
   (insert-3map map loc_atts)
   map))



;;; get_att_locs
;;; get all the attribute locations
(defun get_att_locs (ent_atts)
 (let ((locs))
   (dolist (ent_att1 ent_atts)
            (let* ((ent (first ent_att1))
                     (att (second ent_att1))
                     (loc (get_entity_locs ent att)))
             (setf locs (append locs (insert_ent_att ent_att1 loc)))))
   locs))

;;;;===================
;;;; AUXILIARY FUNCTIONS
;;;;===================

;;; insert_ent_att
;;; inserts (ent att) to each (lqp tb col)
(defmacro insert_ent_att (ent_att locs)
 `(mapcar #'(lambda (x) (append ,ent_att (list x)))
            ,locs))


;;; reverse_map
;;; reverses a list of (ent att (lqp tb col)) into (lqp tb col (ent att))
(defun reverse_map (att_locs)
 (mapcar #'(lambda (x) (append (third x) (list
                                                    (list (first x)
                                                           (second x)))))
          att_locs))
```

APPENDIX C.2 SCHEMA DEFINITION LANGUAGE CODE

```
;;; This file implements the global schema definition language


;;;; CREATE-SCHEMA
(defmacro create-schema (schema)
  "GLOBAL SCHEMA DEFINITION LANGUAGE.
  Creates a global schema.
  Used at the beginning before defining entities and relations
   eg.
    (create-schema alumni)"
  `(progn (create_gsm ',schema)
          (set_current_gsm ',schema)))


;;;; CREATE-ENTITY
(defmacro create-entity (entity &key ((:attributes atts))
                                     ((:table-relations tb-rels)))
  "GLOBAL SCHEMA DEFINITION LANGUAGE.
  Creates an entity with optional attributes and table relations.
  eg. (create-entity 'alumni
                 :attributes
                ((name (lqp1 db1 name1) (lqp2 db2 name2))
                 (age (lqp2 db2 age2)))
                 :table-relations
                ((merge (lqp1 db1) (lqp2 db2)
                 on (= (lqp1 db1 name1) (lqp2 db2 name2)))))"
  `(progn
    (create_entity ',entity)
    (add_gsm_entity ',*current-gsm* ',entity)
    ;; add entity name to current gsm
     (if ',atts
         (progn (dolist (att ',atts)
                        (let ((attribute (first att))
                              (locs (rest att)))
                         ;; add attribute to entity
                         (add_attribute ',entity attribute)
                         (dolist (loc locs)
                                 (add_location ',entity attribute loc))))))
    (if ',tb-rels
        (dolist (rel ',tb-rels)
               (add_table_relation ',entity rel)))))



;;;; CREATE-RELATION
(defmacro create-relation (relation &key ((:entity-from ent_from))
                                          ((:entity-to ent_to)) ((:join join)))
  "Creates a relation.
   eg. (create-relation 'works_for
                   :entity-from 'alumni
                   :entity-to 'company
                   :join '(= (alumni comp-name)
                            (company name)))"
  `(progn
    (create_relation ',relation)
  ;; add relation name to current gsm
  (add_gsm_relation ',*current-gsm* ',relation)
   ;; add to relation object slots
   (if ',ent_from
         (progn
          (add_relation_entity_from ',relation ',ent_from)
          ;; add relation name to entities
          (add_relation ',ent_from ',relation)))
   (if ',ent_to
         (progn
          (add_relation ',ent_to ',relation)
          (add_relation_entity_to ',relation ',ent_to)))
   (if ',join
         (add_relation_join ',relation ',join))))


;;; ================
;;; DELETING A SCHEMA
;;; ================

;;;; delete-schema
;;; deletes an entire schema
(defmacro delete-schema ()
 "GLOBAL SCHEMA DEFINITION LANGUAGE
  Deletes a schema if specified, else deletes the current schema.
  eg.
    (delete-schema) or (delete-schema mit-placement)"
 `(if ',*current-gsm*
     (let ((ents (get_gsm_entities ',*current-gsm*))
             (rels (get_gsm_relations ',*current-gsm*)))
          ;; delete entities
          (dolist (ent1 ents)
                    (delete_entity ent1))
          ;; delete relations
          (dolist (rel1 rels)
                    (delete_relation rel1))
          ;; delete gsm
          (delete_gsm ',*current-gsm*)
          ;; set current gsm to nil
          (set_current_gsm nil))
   nil))


;;;; LOAD-SCHEMA
;;; loads a schema specified by a filename
(defmacro load-schema (file)
 `(if (open ,file :direction :probe)
     (progn
          (delete-schema)
          (load ,file)
          (format t "Current schema ~A~%" ',*current-gsm*)
          t)
   (format "~%File not found: ~A~%" ,file)))
```

```
;;;; VIEW-SCHEMA
;;; views the entities and relations in the schema
(defmacro view-schema ()
 `(if ',*current-gsm*
     (progn
          (format t "~2%GLOBAL SCHEMA: ~A~%" ',*current-gsm*)
          (format t "  Entities: ~%")
          (format t "~5T~A~%" (get_gsm_entities ',*current-gsm*))
          (format t "  Relations: ~%")
          (format t "~5T~A~2%" (get_gsm_relations ',*current-gsm*)))
    (progn
     (format t "~%No schema currently loaded~%")
     (format t "Load schema with (load-schema <filename>)~%"))))


;;;; VIEW-ENTITY
;;; views the attributes, locations, table relations in the entity
(defmacro view-entity (entity)
 "GLOBAL SCHEMA DEFINITION LANGUAGE
  Shows a entity
  e.g.
    (view-entity alumni)"
 `(if (entity_exist? ',*current-gsm* ',entity)
     (let ((atts (get_entity_attributes ',entity)))
          (format t "~2%ENTITY: ~A~%" ',entity)
          (format t "  Attributes:~%")
          (format t "~5T~A~%" atts)
          (format t "  Relations:~%")
          (format t  "~5T~A~%" (get_entity_relations ',entity))
          (format t "  Table-relations:~%")
          (format t "~5T~A~%" (get_entity_table_rels ',entity))
          (dolist (att atts)
                    (format t "  ~A:~%" att)
                    (format t "~5T~A~%"  (get_entity_locs ',entity att)))
          (format t "~%"))
    (progn
     (format t "Entity not found: ~A~%" ',entity)
     (format t "Entities found in global schema:~%")
     (format t "~5T~A~%" (get_gsm_entities ',*current-gsm*)))))


;;;; VIEW-LOCATION
;;; look at the locations of a specific attribute


;;;; VIEW-RELATION
;;; views a relation
(defmacro view-relation (relation)
 "GLOBAL SCHEMA DEFINITION LANGUAGE
  Shows a relation.
  e.g.
   (view-relation works_for)"
 `(if (relation_exist? ',*current-gsm* ',relation)
     (progn
          (format t "~2%RELATION: ~A~%" ',relation)
          (format t "  Entity-from:~%")
          (format t "~5T~A~%" (get_relation_entity_from ',relation))
```

```
        (format t "  Entity-to:~%")
        (format t "~5T~A~%" (get_relation_entity_to ',relation))
        (format t "  Join:~%")
        (format t "~5T~A~%~2%" (get_relation_join ',relation)))
  (progn
   (format t "~%Relation not found: ~A~%" ',relation)
   (format t "Relations found in global schema:~%")
  (format t "~5T~A~%" (get_gsm_relations ',*current-gsm*)))))
```

ENTITY MODULE

```
;;; This file implements the entity module
;;; An entity looks like the following:
;;;
;;; slots:
;;; <attribute_name1>:
;;; ...
;;; <attribute_nameN>:
;;; attributes:
;;; |relations|:
;;; |table_rels|:


;;; entity object

(make-object 'entity
             ('superiors 'gsm)
             ('|attributes| t 'multiple-value-f)
             ('|relations| t 'multiple-value-f)
             ('|table_rels| t 'multiple-value-f))


;;; ====================
;;; CREATING AN ENTITY
;;; ====================

;;; create-entity
;;; creating an entity

(defun create_entity (name)
 (create-instance 'entity name))


;;; add_attribute
;;; adds an attribute to an entity

(defun add_attribute (entity att)
 (cond ((attribute_exist? entity att)
        (signal_error 'entity_3 (list entity att)))
       (t
        (put-object entity att t 'multiple-value-f)
        ;; add to attributes slot
        (put-object entity '|attributes| att))))


;;; add-location
;;; adds a location to an attribute

(defun add_location (entity att loc)
 (cond ((loc_exist? entity att loc)
        (signal_error 'entity_5 (list entity att loc)))
       (t
        (put-object entity att loc))))
```

```
;;; add_relation
;;; adds a relation name to the relations slot

(defun add_relation (entity relation)
 (if (entity_relation_exist? entity relation)
     (signal_error 'error (list entity relation))
     (put-object entity '|relations| relation)))


;;; add_table_relation
(defun add_table_relation (entity relation)
 (if (entity_table_relation_exist? entity relation)
     (signal_error 'error_9 (list entity relation))
    (put-object entity '|table_rels| relation)))


;;; ================
;;; GETTING AN ENTITY
;;; ================


;;; get_entity_table_rels
;;; gets the table relations in the table_rels slot

(defun get_entity_table_rels (entity)
 (get_entity_slot entity '|table_rels|))


;;; get_entity_attributes
;;; gets the attributes in the entity

(defun get_entity_attributes (entity)
 (get_entity_slot entity '|attributes|))


;;; get_entity_relations
;;; gets the relations in the entity

(defun get_entity_relations (entity)
 (get_entity_slot entity '|relations|))


;;; get_entity_locs
(defun get_entity_locs (entity att)
 (get-object entity att))

;;; ================
;;; KOREL INTERFACE
;;; ================

;;; get_entity_slot
;;; gets a slot from an entity object

(defun get_entity_slot (entity slot)
 (get-object entity slot))
```

```
;;; ==============================
;;; DELETING AN AN ENTITY
;;; ==============================

;;; delete_entity
(defun delete_entity (entity)
 (remove-classes entity)
 (remove-object 'entity 'instances entity))


;;; ==============================
;;; AUXILIARY FUNCTIONS
;;; ==============================

;;; attribute_exist?
;;; checks whether attribute exist
(defun attribute_exist? (entity att)
  (let ((atts (get_entity_slot entity '|attributes|)))
   (if (member-equal att atts)
          t
     nil)))


;;; loc_exist?
(defun loc_exist? (entity att loc)
  (if (attribute_exist? entity att)
     (let ((locs (get_entity_slot entity att)))
          (if (member-equal loc locs)
             t
            nil))
   (signal_error 'entity_4 (list entity att))))


;;; checks whether the relation exist
(defun entity_relation_exist? (entity relation)
  (let ((rels (get_entity_slot entity '|relations|)))
   (if (member-equal relation rels)
          t
     nil)))

;;; entity_table_relation_exist?
(defun entity_table_relation_exist? (entity tb-rel)
 (let ((tb-rels (get-object entity '|table_rels|)))
   (if (member-equal tb-rel tb-rels)
          t
     nil)))
```

RELATION MODULE

```
;;; This file implements the relation object and its operations
;;;  A relation looks like the following:
;;;
;;; slots:
;;; entity_to:
;;; entity_from:
;;; join:


;;; relation object

(make-object 'relation
             ('superiors 'gsm)
             ('|entity_to| nil 'multiple-value-f)
             ('|entity_from| nil 'multiple-value-f)
             ('|join| nil 'multiple-value-f))


;;;==================
;;; CREATE A RELATION
;;;==================

;;;; create_relation
(defun create_relation (name)
 (create-instance 'relation name))


;;;; add_relation_entity_from
(defun add_relation_entity_from (relation ent_from)
 (cond ((relation_entity_exist? relation '|entity_from| ent_from)
        (signal_error 'error_8 (list relation 'entity_from ent_from)))
       (t
        (put-object relation '|entity_from| ent_from))))


;;;; add_relation_entity_to
(defun add_relation_entity_to (relation ent_to)
 (cond ((relation_entity_exist? relation '|entity_to| ent_to)
        (signal_error 'error_8 (list relation 'entity_to ent_to)))
       (t
        (put-object relation '|entity_to| ent_to))))


;;; add_relation_join
(defun add_relation_join (relation join)
 (cond ((relation_join_exist? relation join)
        (singal_error 'error_9 (list relation join)))
       (t
        (put-object relation '|join| join))))
```

```
;;; ====================
;;; GETTING A RELATION
;;; ====================

;;; get_relation_entity_from
;;; gets the first entity

(defun get_relation_entity_from (relation)
  (get-object relation '|entity_from|))


;;; get_relation_entity_to
;;; gets the entity_to slot

(defun get_relation_entity_to (relation)
  (get-object relation '|entity_to|))


;;; get_relation_join
;;; gets the join slot

(defun get_relation_join (relation)
  (get-object relation '|join|))


;;; ====================
;;; deleting a relation
;;; ====================

;;; delete_relation
(defun delete_relation (relation)
  (remove-classes relation)
  (remove-object 'relation 'instances relation))


;;; ====================
;;; AUXILIARY FUNCTIONS
;;; ====================

;;; relation_entity_exist?
(defun relation_entity_exist? (rel slot ent)
  (let ((ent_from_slot (get-object rel slot)))
    (if (equal ent ent_from_slot)
          t
      nil)))


;;; relation_join_exist?
(defun relation_join_exist? (rel join)
  (let ((rel_join (get-object rel '|join|)))
    (if (equal join rel_join)
          t
      nil)))
```

GSM MODULE

```
;;;; This files implements the global schema manager (gsm)
;;; The gsm object looks like the following:
;;;
;;; slots:
;;; entities:
;;; relations:

;;; GSM object
(make-object 'gsm
             ('|entities| t 'multiple-value-f)
             ('|relations| t 'multiple-value-f))

;;;; current gsm
(defvar *current-gsm* nil)

;;; set the current gsm
(defun set_current_gsm (gsm)
 (setf *current-gsm* gsm))

;;; ==============
;;; CREATING A GSM
;;; ==============

;;; create_gsm
;;; only one gsm is assumed to be loaded in at any one time. Thus,
;;; there is no need to check whether any other gsm exists.
(defun create_gsm (name)
 (create-instance 'gsm name))

;;; add_gsm_entity
;;; adds an entity name to the entities slot of the gsm
(defun add_gsm_entity (gsm entity)
 (cond ((entity_exist? gsm entity)
          (signal_error 'error_10 (list gsm entity)))
         (t
          (put-object gsm '|entities| entity))))

;;;  adds a relation name to the relations slot of the gsm
(defun add_gsm_relation (gsm relation)
 (cond ((relation_exist? gsm relation)
          (signal_error 'error_11 (list gsm relation)))
         (t
          (put-object gsm '|relations| relation))))


;;; ==========
;;; GETTING GSM
;;; ==========

;;; get_gsm_relations
(defun get_gsm_relations (gsm)
 (get-object gsm '|relations|))

;;; get_gsm_entities
(defun get_gsm_entities (gsm)
 (get-object gsm '|entities|))
```

```
;;; ============
;;; DELETE A GSM
;;; ============

;;; delete_gsm
(defun delete_gsm (gsm)
 (remove-classes gsm)
 (remove-object 'gsm 'instances gsm))

;;; delete_gsm_entity
;;; delete an entity in the entities slot
(defun delete_gsm_entity (gsm entity)
 (remove-object gsm '|entities| entity))


;;; delete_gsm_relation
;;; delete a relation in the relations slot
(defun delete_gsm_relation (gsm relation)
 (remove-object gsm '|relations| relation))


;;; ===================
;;; AUXILIARY FUNCTIONS
;;; ===================

;;; entity_exist?
(defun entity_exist? (gsm entity)
 (let ((gsm_ents (get-object gsm '|entities|)))
  (if (member-equal entity gsm_ents)
       t
    nil)))

;;; relation_exist?
(defun relation_exist? (gsm relation)
 (let ((gsm_rels (get-object gsm '|relations|)))
  (if (member-equal relation gsm_rels)
       t
    nil)))
```

```
;;;; This files implements the global schema manager (gsm)
;;; The gsm object looks like the following:
;;;
;;; slots:
;;; entities:
;;; relations:

;;; GSM object
(make-object 'gsm
             ('|entities| t 'multiple-value-f)
             ('|relations| t 'multiple-value-f))

;;;; current gsm
(defvar *current-gsm* nil)

;;; set the current gsm
(defun set_current_gsm (gsm)
 (setf *current-gsm* gsm))

;;; ==============
;;; CREATING A GSM
;;; ==============

;;; create_gsm
;;; only one gsm is assumed to be loaded in at any one time. Thus,
;;; there is no need to check whether any other gsm exists.
(defun create_gsm (name)
 (create-instance 'gsm name))

;;; add_gsm_entity
;;; adds an entity name to the entities slot of the gsm
(defun add_gsm_entity (gsm entity)
 (cond ((entity_exist? gsm entity)
          (signal_error 'error_10 (list gsm entity)))
         (t
          (put-object gsm '|entities| entity))))

;;;  adds a relation name to the relations slot of the gsm
(defun add_gsm_relation (gsm relation)
 (cond ((relation_exist? gsm relation)
          (signal_error 'error_11 (list gsm relation)))
         (t
          (put-object gsm '|relations| relation))))


;;; ============
;;; GETTING GSM
;;; ============

;;; get_gsm_relations
(defun get_gsm_relations (gsm)
 (get-object gsm '|relations|))

;;; get_gsm_entities
(defun get_gsm_entities (gsm)
 (get-object gsm '|entities|))
```

```
;;; ============
;;; DELETE A GSM
;;; ============

;;; delete_gsm
(defun delete_gsm (gsm)
 (remove-classes gsm)
 (remove-object 'gsm 'instances gsm))

;;; delete_gsm_entity
;;; delete an entity in the entities slot
(defun delete_gsm_entity (gsm entity)
 (remove-object gsm '|entities| entity))


;;; delete_gsm_relation
;;; delete a relation in the relations slot
(defun delete_gsm_relation (gsm relation)
 (remove-object gsm '|relations| relation))


;;; ===================
;;; AUXILIARY FUNCTIONS
;;; ===================

;;; entity_exist?
(defun entity_exist? (gsm entity)
 (let ((gsm_ents (get-object gsm '|entities|)))
  (if (member-equal entity gsm_ents)
        t
    nil)))

;;; relation_exist?
(defun relation_exist? (gsm relation)
 (let ((gsm_rels (get-object gsm '|relations|)))
  (if (member-equal relation gsm_rels)
        t
    nil)))
```

APPENDIX C.3 GLOBAL QUERY LANGUAGE CODE

```
;;; This implements the global query language

;;; =========
;;; condition
;;; =========

;;; make_simple_cond
;;; makes a simple condition

(defun make_simple_condition (op att val)
 (list op att val))


;;; make_nested_cond
;;; makes a nested condition

(defun make_nested_cond (op cond1 cond2)
 (list op cond1 cond2))


;;; simple_cond?
;;; checks whether it is a simple condition, ie. and operator or an "in".

(defun simple_cond? (conds)
 (if (or (in_cond? conds)
         (op_cond? conds))
    t
   nil))


;;; nested_cond?
;;; checks whether the condition is an and or or operator

(defun nested_cond? (conds)
 (if (or (and_cond? conds)
         (or_cond? conds))
    t
   nil))


;;; and_cond?
;;; checks whether it is an AND condition

(defun and_cond? (cond)
 (if (equal (first cond) 'and)
    t
   nil))


;;; or_cond?
;;; checks whether it is an OR condition

(defun or_cond? (cond)
 (if (equal (first cond) 'or)
    t
   nil))


;;; in_cond?
;;; checks whether it is an IN condition

(defun in_cond? (cond)
 (if (equal (first cond) 'in)
    t
   nil))


;;; op_cond?
;;; checks whether it is an OPERATOR condition

(defun op_cond? (cond)
 (if (member-equal (first cond) '(= > < >= <=))
    t
   nil))


;;; ===============
;;; JOIN statement
;;; ===============

;;; make_join
;;; make a join statement

(defun make_join (join1 join2 join_on)
 (list 'join join1 join2 'on join_on))


;;; join?
;;; checks whether the query is a join statement

(defun join? (query)
 (if (equal (first query) 'join)
    t
   nil))


;;; get_join1
;;; get the first part of the join

(defun get_join1 (query)
 (second query))


;;; get_join2
;;; get the second part of the join

(defun get_join2 (query)
 (third query))
```

```
;;; get_join_on
;;; get the on information

(defun get_join_on (query)
 (fifth query))

;;; relation_join?
;;; checks whether the on part of the join statement has a relation
;;; specified in terms of entities, eg. (= (alumni name) (company
;;; employee)) *** should have a better way to check**
(defun relation_join? (join)
 (cond ((atom join) nil)
          ((join_op? join)
           t)
          (t
           nil)))


;;; join_op?
;;; checks the first thing in the list for the join operator
(defun join_op? (join)
 (if (or (join_=? join) (join_and? join))
     t
   nil))

;;; join_=?
(defun join_=? (join)
 (if (equal (first join) '=)
     t
   nil))


;;; join_and?
;;; checks whether it is an and
(defun join_and? (join)
 (if (equal (first join) 'and)
     t
   nil))

;;; gets the first and of a condition
(defun join_and1 (cond)
 (second cond))

;;; gets the second and of a condtion
(defun join_and2 (cond)
 (third cond))


;;; ================
;;; SELECT statement
;;; ================


;;; make_select
;;; make a select statement

(defun make_select (entity atts conds)
 (let ((sel_query (list 'select entity atts)))
   (if conds
         (setf sel_query (append sel_query (list 'where conds))))
   sel_query))


;;; select?
;;; checks whether the query is a select statement

(defun select? (query)
 (if (equal (first query) 'select)
     t
   nil))


;;; get_select_entity
;;; get the entity

(defun get_select_entity (query)
 (second query))


;;; get_select_atts
;;; gets the attribute list in a select statement

(defun get_select_atts (query)
 (third query))


;;; get_select_conds
;;; gets the condition list

(defun get_select_conds (query)
 (fifth query))


;;; *-option?
;;; returns true if the attribute list is a *

(defun *-option? (op)
 (if (equal op '*)
     t
   nil))
```

APPENDIX C.4 INTERMEDIATE QUERY LANGUAGE MODULES

```
;;;; This file implements the intermediate query
;;; April 16, 89

;;; GET-TABLE

;;; make_table
(defun make_table (lqp tb atts &optional conds)
 (if conds
     (list 'get-table lqp tb atts 'where conds)
   (list 'get-table lqp tb atts)))


;;; get-table?
;;; checks whether the query is a get-table statement
(defun get-table? (query)
 (if (equal (first query) 'get-table)
     t
   nil))


;;; get_table_lqp
;;; gets the lqp part
(defun get_table_lqp (query)
 (second query))


;;; get_table_tb
;;; gets the table part
(defun get_table_tb (query)
 (third query))


;;; get_table_atts
;;; gets the list of attributes
(defun get_table_atts (query)
 (fourth query))


;;; get_table_conds
(defun get_table_conds (query)
 (nth 5 query))


;;; make_table_or
(defun make_table_or (or1 or2)
 (list 'or or1 or2))

;;; make_table_and
(defun make_table_and (and1 and2)
 (list 'and and1 and2))

;;;; MERGE

;;; make_merge
(defun make_merge (m1 m2 on)
  (list 'merge m1 m2 'on on))


;;; get_merge1
;;; get m1
(defun get_merge1 (query)
  (second query))


;;; get_merge2
;;; get m2
(defun get_merge2 (query)
  (third query))

;;; merge?
;;; checks whether it is a merge statement
(defun merge? (query)
  (if (equal (first query) 'merge)
      t
    nil))


;;; get_merge_on
(defun get_merge_on (query)
  (nth 4 query))

;;; merge_and?
;;; checks whether the on part
(defun merge_and? (merge_on)
  (if (equal (first merge_on) 'and)
      t
    nil))

;;; merge_and1
;;; gets the first part of an and
(defun merge_and1 (merge_on)
  (second merge_on))

;;; merge_and2
(defun merge_and2 (merge_on)
  (third merge_on))


;;; CONCATENATE

;;; make_concatenate
(defun make_concatenate (c1 c2)
  (list 'concatenate c1 c2))

;;; concatenate?
(defun concatenate? (query)
  (if (equal (first query) 'concatenate)
      t
    nil))

;;; get_concatenate1
(defun get_concatenate1 (query)
  (second query))
```

```
;;; get_concatenate2
(defun get_concatenate2 (query)
 (third query))


;;; =====
;;; merge
;;; =====

;;; eg. (MERGE (db1 tb1) (db1 tb2)
;;;        ON (AND (= (db1 tb1 name) (db1 tb2 name1))
;;;              (= (db1 tb1 first_name) (db1 tb2 first_name1))))


;;; merge?
;;; checks whether it is a merge statement

(defun merge? (join)
 (if (equal (first join) 'merge)
     t
   nil))


;;; and_join?
;;; description: returns t if it is an "and" join

(defun and_join? (join)
 (if (equal (first join) 'and)
     t
   nil))


;;; simple_join?
;;; description: returns t if it is a simple join, ie. (= ....)

(defun simple_join? (join)
 (if (equal (first join) '=)
     t
   nil))


;;; get_jtb1
;;; description: get the first db-table from the join information

(defun get_jtb1 (join)
 (second join))


;;; get_jtb2
;;; description: get the second db-table from the join information

(defun get_jtb2 (join)
 (third join))
```