# Multibase—integrating heterogeneous distributed database systems*

*by* JOHN MILES SMITH, PHILIP A. BERNSTEIN, UMESHWAR DAYAL, NATHAN GOODMAN, TERRY LANDERS, KEN W. T. LIN, and EUGENE WONG

*Computer Corporation of America*
Cambridge, Massachusetts

## ABSTRACT

Multibase is a software system for integrating access to preexisting, heterogeneous, distributed databases. The system suppresses differences of DBMS, language, and data models among the databases and provides users with a unified global schema and a single high-level query language. Autonomy for updating is retained with the local databases. The architecture of Multibase does not require any changes to local databases or DBMSs. There are three principal research goals of the project. The first goal is to develop appropriate language constructs for accessing and integrating heterogeneous databases. The second goal is to discover effective global and local optimization techniques. The final goal is to design methods for handling incompatible data representations and inconsistent data. Currently the project is in the first year of a planned three year effort. This paper describes the basic architecture of Multibase and identifies some of the avenues to be taken in subsequent research.

## 1. INTRODUCTION

### *What is Multibase?*

The database approach to data processing requires that all of the data relevant to an enterprise be stored in an integrated database. By "integrated," we mean that a single schema (i.e., database description) describes the entire database, that all accesses to the database are expressed relative to that schema, and that such accesses are processed against a single (logical) copy of the database. Unfortunately, in the real world many databases are not integrated. Often, the data relevant to an enterprise is implemented by many independent databases, each with its own schema. Such databases are nonintegrated. Furthermore, these databases may be managed by different database management systems (DBMS), perhaps on different hardware. In this case, in addition to being nonintegrated the databases are distributed and heterogeneous. Thus, the real world of nonintegrated, heterogeneous, distributed databases differs greatly from the more ideal world of an integrated database.

Nonintegrated, heterogeneous, distributed databases arise for several reasons. First, many of these databases were created before the benefits of integrated databases were well understood. In those days, total integration was not a principal database design goal. Second, the lack of a central database administrator for some enterprises has made it difficult for independent organizations within an enterprise to produce an integrated database suitable for all of them. Third, the large size of many data processing applications has made distribution a necessity, simply to handle the volume of work. Since integrated distributed DBMSs have not been available, it has been necessary to implement applications on different machines. Since different applications often have different performance and functionality requirements, different DBMSs were often selected to run on these machines to meet these different requirements. Many data processing organizations have experienced these problems, so there are many nonintegrated, heterogeneous, distributed databases in the world.

A principal problem in using databases of this type is that of integrated retrieval. In such databases, each independent database has its own schema, expressed in its own data model, and can be accessed only by its own retrieval language. Since different databases in general have different schemata, different data models, and different retrieval languages, many difficulties arise in formulating and implementing retrieval requests (called queries) that require data from more than one database. These difficulties include the following: resolving incompatibilities between the databases, such as differences of data types and conflicting schema names; resolving inconsistencies between copies of the same information stored in different databases; and transforming a query expressed in the user's language into a set of queries expressed in the many different languages supported by the different sites. Implementing such a query usually consumes months of program-

* This research was jointly supported by the Defense Advanced Research Projects Agency of the Department of Defense and the Naval Electronic Systems Command and was monitored by the Naval Electronic Systems Command under Contract No. N00039-80-C-0402. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Naval Electronic Systems Command or the U.S. Government.

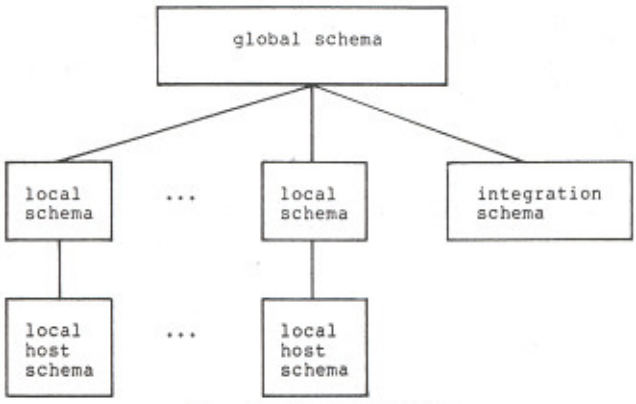


Figure 1—Schema architecture

ming time, making it a very expensive activity. Sometimes, the necessary effort is so great that implementing the query is not feasible at all.

Multibase is a software system that helps integrate non-integrated, heterogeneous, distributed databases. Its main goal is to present the illusion of an integrated database to users without requiring that the database be physically integrated. It accomplishes this by allowing users to view the database through a single global schema and by allowing them to access the data using a high level query language. Queries posed in this language are entirely processed by Multibase as if the database were integrated, homogeneous, and non-distributed. Multibase uses the Functional Data Model[1] to define the global schema, and the language DAPLEX[1] as the high level query language.

*Implementation Objectives*

There are many approaches to the design of the Multibase system. In deciding which approach to choose, we begin with the following design objectives.

1. Generality: we do not want to design an application-specific Multibase system. Instead, we want to provide powerful generalized tools that can be used to integrate various database systems for various applications with a minimum of programming effort.
2. Extendability: we want a design that allows expansion of functionality without major modification. There are areas in the Multibase design where substantial research effort is still required, so we must be able to add additional features to the Multibase system as we learn more about the problems.
3. Compatibility: we want a design that does not render existing software invalid, because such software represents a very large investment. Thus, we must leave the existing interface to the local DBMS intact.

The proposed architecture of the Multibase system consists of two basic components: a schema design aid and a run-time query processing subsystem. The schema design aid provides tools to the "integrated" database designer to design the global schema and to define a mapping from the local databases to the global schema. The run-time query processing subsystem then uses the mapping definition to translate global queries into local queries, ensuring that the local queries are executed correctly and efficiently by local DBMSs. The schema design aid is discussed first.

*Schema Architecture*

The Multibase architecture has three levels of schemata, a global schema (GS) at the top level, an integration schema (IS) and one local schema (LS) per local database at the middle level, and one local host schema (LHS) per local database at the bottom level. These components and their interrelationships are depicted in Figure 1.

The local host schemata are the original existing schemata defined in local data models and used by the local DBMSs. For example, they can be relational, file, or CODASYL schemata. Each of these LHSs is translated into a local schema (LS) defined in the Functional Data Model. By expressing the LSs in a single data model, higher levels of the system need not be concerned with data model differences among the local DBMSs. In addition, there is an integration schema that describes a database containing information needed for integrating databases. For example, suppose one database records the speed of ships in miles per hour, while the other records it in kilometers per hour. To integrate these two databases, we need information about the mapping between these two scales. This information is stored in the integration database.

The LSs and IS are mapped, via a view mapping, into the global schema (GS). The GS allows users to pose queries against what appears to be a homogeneous and integrated database. Roughly speaking, the LHS to LS mapping provides homogeneity and the LS and IS to GS mapping provides integration. The schema design aid provides tools to the database designer to define LSs, the GS, and the mapping among them and the LHSs.

*Query Processing Architecture*

The architecture of the run-time query processing subsystem consists of the Multibase software and local DBMSs.

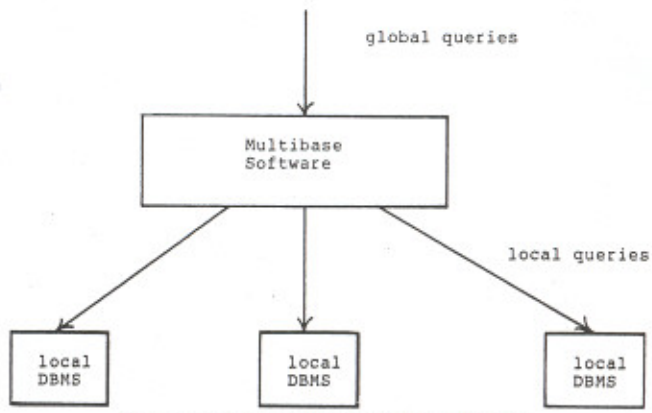


Figure 2—Run-time query processing subsystem

These components and their interrelationships are depicted in Figure 2. The users submit queries over the global schema (called global queries) to the Multibase software, which translates them into subqueries over local schemata (called local queries). These local queries are then sent to local DBMSs to be executed.

Since the global queries are posed against the global schema without any knowledge of the distribution of the data and the availability of "fast access paths," the Multibase software must optimize queries so they can be executed efficiently. In addition, the translation process must also be correct; that is, the local queries must retrieve exactly the information that the original global query requests.

*Meeting the Objectives*

The proposed architecture meets the objective of generality. The only component of the Multibase system that is customized for the application is the global schema and its mapping definition to the local schemata. The only component of Multibase that is customized for the local DBMSs is the interface software that allows Multibase to communicate with the heterogeneous DBMSs in a single language. These are only small components of the Multibase system. Thus, most of Multibase is neither application-specific nor DBMS-specific. Multibase also meets the objective of compatibility, because local databases are not modified; therefore, existing application programs can still access local databases through local DBMSs. And as the details of the architecture are discussed in later sections, it will become clear that the objective of extendability is also met.

*Project Status*

The Multibase project is a three-year effort. Within the first two years, the research problems in the system design will be resolved and evaluated, using a "breadboard" implementation of the system. In the final year, a revised design will be developed and implemented in ADA. The ADA version will be made available for experimental testing within the Navy "Command and Control" environment.

It is anticipated that the major research problems are

1. basic architecture of the system,
2. global and local optimization, and
3. handling incompatible data.

At the time of this writing, an architecture has been designed that supports a restricted version of DAPLEX with reasonable efficiency and that can be tailored to handle certain kinds of data incompatibility. This basic architecture is currently being implemented as a breadboard system. Subsequently, research will be devoted to removing the restrictions on DAPLEX and investigating algorithms for processing incompatible data. The breadboard system will then be enhanced to include the new capabilities. This paper describes the basic architecture developed to date.

*Organization*

The architecture of the Multibase system is expanded in more detail in Section 2. The process of mapping each LHS to a LS and merging LSs into a GS is discussed in Section 3. Section 3 also discusses the problem of data incompatibility and inconsistency. The method by which user queries are translated into efficient local queries is discussed in Section 4. Section 5 is a summary.

## 2. QUERY PROCESSING ARCHITECTURE

The architecture of the Multibase run-time subsystem consists of

1. a query translator,
2. a query processor,
3. a local database interface (LDI) for each local DBMS, and
4. local DBMSs.

A global query references entity types and functions defined in the global schema. Before it can be processed, it must be translated by the query translator into a query referencing only entity types and functions defined in the local schemata. In other words, the query translator translates a global query over the global schema into a global query over the disjoint union of local schemata. The query processor decomposes the global query over the disjoint union of local schemata into individual local queries over local schemata. The query processor also does query optimization and coordinates the execution of local queries. The LDI translates local queries received from the query processor into queries expressed in the local DML and translates the results of the local queries into a format expected by the query processor. These components and their interrelationships are depicted in Figure 3.

*The User Interface*

The global schema is expressed in the functional data model.[1] In this data model, a schema is composed of *entity types* and *functions* between entity types. Each entity type contains a set of entities, so functions map entities into entities. Functions can be *single-valued* or *multi-valued,* and can be *partially defined* or *totally defined.*

The functional data model was selected because it embodies the main structures of both the flat file data models, such as the relational model, and the link structured data models, such as CODASYL. Entity types correspond roughly to relations in the relational model or record types in the CODASYL model. Functions correspond to owner-coupled sets in the CODASYL model.

The query language that we use with the functional data model is called DAPLEX. DAPLEX is a high level language that operates on data in the functional data model and is designed to be especially easy to use by end users.

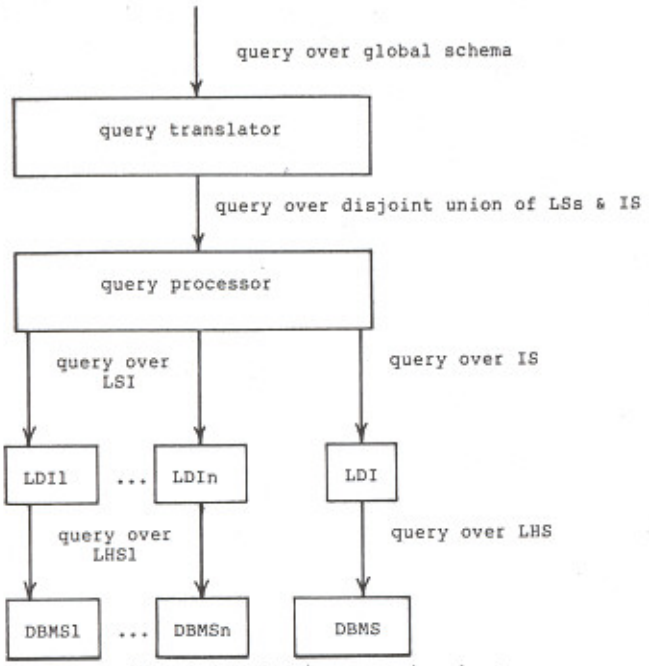


Figure 3—Run-time query processing subsystem

*Query Translator*

The query translator receives global queries expressed in DAPLEX over the GS and translates them into queries expressed in an internal language over the disjoint union of LSs and IS.

To perform the translation, the query translator must use the mapping that defines how entity types and functions of the GS are constituted from the entity types and functions of the LS and the IS. The query translator uses these mapping definitions to substitute global entity types and global functions in the global query by their mapping definitions. The substitution results in a query containing only entity types and functions of the LSs and the IS. Therefore references by the global query to entities in the GS are now expressed as references to the actual entities at particular sites that implement the global GS. Any extra data needed from the integration database to resolve incompatibilities among LSs is now explicitly referenced in the translated query.

The query produced by the query translator only references data in the LS and the IS. Thus, we can imagine that this query is posed against a database state that is the disjoint union of the LSs together with the IS. This disjoint union is a homogeneous and centralized view of the distributed heterogeneous database.

The language used for defining the mapping between schemata must be compatible with the global DML. Otherwise, it would be awkward to translate the query from the GS to LSs and IS using conventional query modification techniques. (Query modification composes the given query, which is a function from GS states to answer states, with the mapping from LS and IS states to GS states, to produce a query from LS and IS states to answer states.[2]) Therefore, we propose to use the same language DAPLEX as both the query and mapping language. The process of constructing the global schema from the local schemata is discussed in Section 3.

*Query Processor*

The query processor translates a query over the disjoint union of LSs and IS into a *query processing strategy*. This strategy includes the following: a set of queries, each of which is posed against exactly one LS or the IS; a set of "move" operations to ship the results of these queries between the local DBMSs and the query processor; and a set of queries that is executed locally by the query processor to integrate the results of the LS and IS queries. The main goal of this translation is to minimize the total cost of evaluating the query, where cost is measured by local processing time and communication volume.

A query processing strategy is produced in two steps. First, the query is translated into an internal representation called a *query graph*. Using this representation, the query processor isolates those subqueries of the given query (which are essentially subgraphs of the query graph) that can be entirely evaluated at one local DBMS. Thus, the result of the first step is the set of single-site subqueries of the given query.

The second step is to combine the single-site queries with move operations and local queries issued by the query processor. Move operations serve two purposes. First, they are used to gather the results of the single-site queries back to the query processor. These results can be integrated by the query processor by executing a query local to itself. The integrated results may be the answer to the query, in which case they are returned to the user. Second, they may be used as input to other single-site queries. In this case, a move operation is issued to ship the data to the local DBMS that needs it. The method by which single-site queries, move operations, and queries local to the query processor are sequenced to produce a correct and efficient strategy is discussed in Section 4.

*Local Database Interface (LDI)*

Local queries posed against the LSs are sent by the query processor to the LDIs in an internal format. The LDI translates these local queries into programs in the local DML and programming language over the local host schema (LHS). This translation is optimized to minimize the processing time of the translated query. When the local DBMS uses a high level (i.e., set-at-a-time) language, such as DAPLEX, this translation is fairly direct. However, when the local DBMS uses a low level (i.e., record-at-a-time) language, such as CODASYL DML embedded in COBOL, this translation may be quite complex and may require nontrivial optimization. Translation methods for a file system and CODASYL language are described in Section 4.

To do the translation, the LDI must have information about how entity types and functions in the LS are mapped to objects in the LHS. These mappings are defined using the rules discussed below.

## 3. SCHEMA INTEGRATION ARCHITECTURE

"Schema Integration" is the process of defining a global schema and its mapping from the existing local schemata. The general architecture of this design process is discussed in this section.

There is one local host schema (LHS) for each local database. Each LHS can be expressed in a relational, CODASYL, or a file language. To merge these LHSs we must convert them into a common data model first. Otherwise, we would be mixing relations from a relational model with record types and set types from a CODASYL model. Thus the first step of schema integration is to translate LHSs into Local Schemata (LS) defined in the Functional Data Model of DAPLEX.

The second step is to merge LSs into a GS. To do this, an integration schema which defines an integration database is often needed. An integration database contains: information about mapping between different scales used by different LSs for the same entity type; statistical information about imprecise data; and other information needed for reconciling inconsistency between copies of the same data stored in different databases. The integration schema and LSs are then used to define a global schema.

The overall architecture of schema integration consists of

a) a global schema,
b) a mapping language,
c) local schemata (LS) and an integration schema (IS),
d) a mechanized local-to-host schema translator, and
e) local host schemata (LHS) and local DBMSs.

These components and their interrelationships are depicted in Figure 4. The local host schemata are translated into local schemata by the mechanized local host schema translator, and local schemata and the IS are mapped into the GS by using the mapping language facility.

### *Mapping between LHS and LS*

Since an LHS can be defined in the relational, CODASYL, or file model, how an LHS is mapped into an LS depends on the data model used.

#### CODASYL model

If an LHS is defined in the CODASYL model, then it consists of record types and set types. The functional data model consists of entity types and functions on entity types. So, to map the LHS into an LS one simply maps record types and set types into entity types and functions respectively.

The concept of record type in the CODASYL model is very similar to that of entity type in the functional data model. A record in the CODASYL model has a record ID, and one or several attributes. The record ID uniquely identifies the record, and the attributes describe properties of the record. Similarly, in the functional data model, an entity is an object of interest, and the functions defined on the entity return values that describe the properties of the entity. Therefore, a record type corresponds to an entity type, and the attributes of the record type correspond to functions defined on the entity type.

If an attribute of a record type is a key (in CODASYL terminology, a key is the data item(s) declared "NO DUPLICATE ALLOWED") then the corresponding function must be a totally defined one-to-one mapping. If the attribute is a repeating group (declared to have multiple occurrences in a CODASYL model), then the corresponding function is a set-valued function.

A set type in the CODASYL model is a mapping between an owner record type and one or several member record types. A set type maps an owner record to a set of member records, or, conversely, a set type maps a member record to a unique owner record. Therefore, a set type resembles a function that maps an owner entity to a set of member entities, or, conversely, maps a member entity to a unique owner entity.

In a CODASYL model, a set type implies not only certain semantic information but also the existence of access paths. For example a set type "work-in" between "department" and "employee" record types implies that the employees owned by a department work in that department. But it also implies that there is an access path from a department record to the employee records owned by that department and another access path from each employee record to its own department record. Since the LSs will be used for query optimization, we

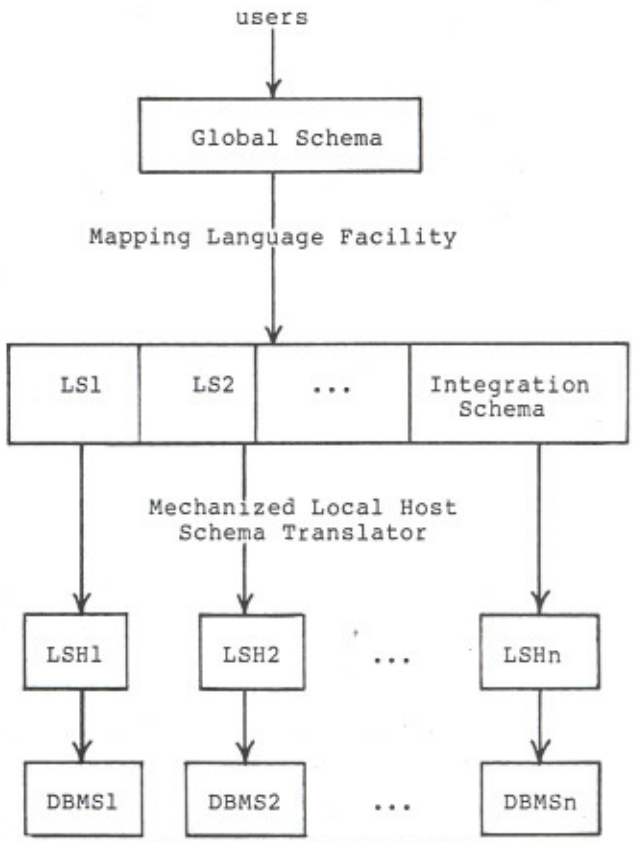
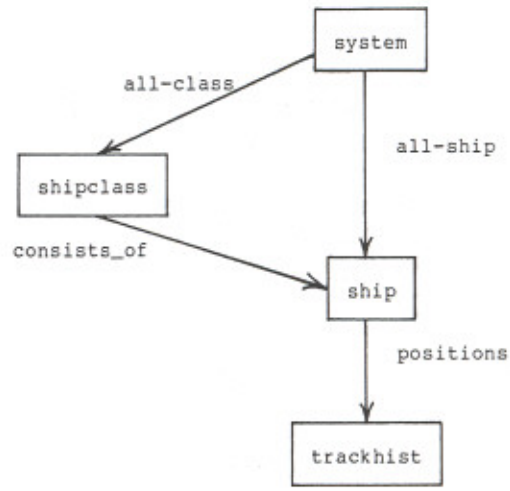



Figure 4—Schema integration architecture

must capture all this access path information in the LSs. Therefore, for each set type in an LHS, not only a set-valued function from the owner entity type to the member entity type, but also a single-valued function from each of the member entity types to the owner entity type must be defined in the corresponding LS.

In a CODASYL model, a record type can be declared to have a "LOCATION MODE CALC USING KEY." This means that an index file is created for the key, and the record type is directly accessible through the indexed key. Therefore, for each record type with "CALC KEY" in the LHS, a system set function of which the domain is the key value and the range is the entity type (corresponding to the record type) must be defined in the LS. This system set function will be used only for query processing optimization. It is not visible to the database designer. Therefore, it cannot be incorporated into the global schema. This restriction is imposed to preserve the data independence of the global schema.

For example, the CODASYL schema shown in Figure 5 is translated into the schema in the functional data model shown in Figure 6. In Figure 6, the inverse of a function F is denoted by "F-inv."




```
Shipclass Record                    Trackhist Record
*classname     char(24)          ** DTG          char(10)
 length        char(6)              speed        char(3)
 draft         char(2)              latitude     char(5)
 beam          char(3)              longitude    char(6)
 displacement  char(5)              course       char(3)
 endurance     char(3)

*  primary key
** key within a set

Ship Record
*  UIC              char(6)
   VCN              char(5)
   name             char(26)
   type             char(4)
   flag             char(2)
   owner            char(2)
   hull             char(4)
```

Figure 5—A CODASYL schema

### Relational model

A relational database schema consists of a set of relation definitions. To translate a relational LHS to a functional LS we essentially map each relation to an entity type. A tuple of a relation in a relational model is similar to an entity in a functional data model. A tuple is uniquely identified by its primary key and has one or more attributes, just as an entity has one or more functional values. Therefore, to map a relational model LHS into a functional data model LS, for each relation in the LHS an entity type is defined in the LS, and for each attribute of the relation a function is defined on the corresponding entity type. The range of the function is the domain of the attribute. If the attribute is a primary key, then the function must be totally defined and one-to-one. If it is a candidate key, then the function can be partially defined, but it must still be one-to-one. In any case, due to the relational format, the function must be single-valued, not set-valued. For example, the relational LHS shown in Figure 7 is translated into the functional data model LS shown in Figure 8.

### File model

A file model consists of record files and indexed fields (keys) in those files. A record file consists of a set of records of the same type, which is similar to the concept of record type in the CODASYL model or a relation in the relational model. To map a file LHS to a functional data model LS, for each record file in LHS a corresponding entity type must be defined in the LS, and for each field of the record file a function must be defined on the entity type. Since a key supports an access path to the record file, for each key of a record file, a system function must be defined whose domain is the key field's entity type and whose range is the entity type corresponding to the record file. This system function is not visible to the database designer; it is used only for query optimization.

*Integration of LSs*

To integrate LSs into a global schema, the database designer designs an integration schema that defines an integration database. He then designs a global schema and defines it in terms of the LSs and the Integration Schema by using the view support facility.

An integration database contains information needed for merging entity types and their functions. For example, two entity types, E1 and E2, from two schemata are shown in Figure 9. These two entity types represent information about ships. There are two functions defined on each entity type; one function returns the ship-id of a ship and the other returns the ship-class of the ship. The ship-class of E1 and E2 are coded differently. A sample of entities and their functional values are also shown in Figure 9. To merge E1 and E2 into a single entity type, a uniform code must be defined, and the two existing codes must be mapped to the new code. Definitions of the new code and the mapping function are shown in Figure 10, and a sample of the function is shown in Figure 11.

```
type shipclass is entity
   classname      : string(1..24);
   length         : string(1..6);
   draft          : string(1..2);
   beam           : string(1..3);
   displacement   : string(1..5);
   endurance      : string(1..3);
   consists-of    : set of ship;
end entity;

type ship is entity
   UIC     : string(1..6);
   VCN     : string(1..5);
   name    : string(1..26);
   type    : string(1..4);
   flag    : string(1..2);
   owner   : string(1..2);
   hull    : string(1..4);
   positions :  set of trackhist;
   consists-of_inv : shipclass;
end entity;


type system is entity
 all-class : set of shipclass
 all-ship  : set of  ship;
end entity;


type trackhist is entity
  DTG       : string (1..10);
  speed     : string (1..3);
  latitude  : string (1..5);
  longitude : string (1..6);
  course    : string (1..3);
  positions_inv : ship;
end entity;
```

Figure 6—A schema in the functional data model

The definitions of the new code and the function are stored in the integration database. A global schema defined on the two local schemata and the integration schema is shown in Figure 12.

As the discussion above indicates, integration of local schemata which are not disjoint involves two activities: merging of entity types and merging of their functions. These activities are discussed in the next section. Two special problems relating to schema integration, the creation of new entity types, and the integration of incompatible data, are discussed in subsequent sections.

```
Relation Platform
   VesselName        char(26)
   class             char(25)
   type              char(6)
   hull              char(6)
   flag              char(2)
   category          char(4)
*  PIF               char(4)
*  NOSICID           char(8)
   IRCS              char(8)

Relation Position
*  PIF               char(4)
*  NOSICID           char(8)
*  DTG               char(10)
   latitude          char(5)
   longitude         char(6)
   bearing           char(3)
   course            char(3)
   speed             char(3)

* primary key
```

Figure 7—A relational model

### *Merging Entity Types and Functions*

To merge two entity types, say E1 and E2 in Figure 9, into an entity type, say E in Figure 12, the database designer must first determine whether the set of entities of type E1 is disjoint from the set of entities of type E2. If E1 and E2 are disjoint, then E is simply the union of E1 and E2. If E1 and E2 are not disjoint, then the condition under which two entities from E1 and E2 respectively are identical must be specified. To specify the condition under which entities are identical, entities of E1 and E2 must be able to be identified by their attributes. Therefore, for each entity type to be merged, a function or combination of functions of the entity type must be a primary key. Two entities from two entity types being merged can then

```
type platform is entity
   VesselName   :string (1..26);
   class        :string (1..25);
   type         :string (1..6);
   hull         :string (1..6);
   flag         :string (1..2);
   category     :string (1..4);
   PIF          :string (1..4);
   NOSICID      :string (1..8);
   IRCS         :string (1..8);
end entity;

type position is entity
   PIF          :string (1..4);
   NOSICID      :string (1..8);
   DTG          :string (1..10);
   latitude     :string (1..5);
   longitude    :string (1..6);
   bearing      :string (1..3);
   course       :string (1..3);
   speed        :string (1..3);
end entity;
```

Figure 8—A schema for the functional data model

```
type E1 is entity
     shipid1 : integer;
     class1  : code1;
end entity;
```

| E1 | shipid1 | class1 |
|---|---|---|
| e11 | 1212 | c1 |
| e12 | 1240 | c3 |
| e13 | 2341 | c5 |

```
type E2 is entity
       shipid2 : integer;
       class2  : code2;
end entity;
```

| E2 | shipid2 | class2 |
|---|---|---|
| e21 | 3440 | d2 |
| e22 | 3651 | d3 |
| e23 | 4411 | d4 |

Figure 9—Local schemata

```
type code is entity
end entity;

Define a new function
     f : (code1 union code2)  -> code.
```

Figure 10—Integration database

Sample of function f

| code1,code2 | c1 | c2 | c3 | c4 | c5 | d1 | d2 | d3 | d4 |
|---|---|---|---|---|---|---|---|---|---|
| code | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Figure 11—Sample of function f

```
type E is entity
     shipid : integer;
     class  : code;
end entity;
```

Figure 12—Global schema

be specified as identical if and only if they have identical primary key values.

In Figure 13, entity types E1 and E2 (which are assumed to overlap), are merged into an entity type E. The syntax used is a subset of DAPLEX. Notice that "shipid1" and "shipid2" are assumed to be primary keys of E1 and E2 respectively. Further, it is assumed that an E1 entity and an E2 entity are identical if and only if they have the same primary key values.

*Creation of a New Entity Type and its Functions*

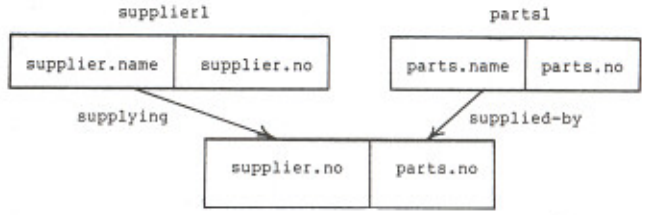
Merging two entity types into a single entity type is a special case of creating a new entity type. Essentially, a new entity type may be created which is a combination of the existing entity types. However, this combination does not create new objects in the database. Rather, it simply presents many existing objects of different types as objects of a single type to the global schema users. Properties of the new global entities are simply those that previously existed in the local schemata.

However, in some cases, a database designer may want to design a more sophisticated global schema in which new (virtual) objects derive their properties (attributes) from many dissimilar existing objects. An example is used to illustrate this process, and general principles can be drawn from the example.

```
type E is entity
   shipid : integer;
   class  : code;
end entity;

for each  x in E1 where not (shipid1(x) isin
                             shipid2(E2))
loop
create new E(shipid => shipid1(x)
             class  => f (class1(x)));
end loop;

for each x in E2

loop

create new E(shipid => shipid2(x),

             class  => f(class2(x)));

end loop;
```

Figure 13—The mapping definition of entity type E

Local Schema 1:




```
type supplier1 is entity
     sname      : string(30);
     sno        : integer;
     supplying  : set of supply1;
end entity;

type parts1 is entity
     pname      : string(15);
     pno        : integer;
     supplied-by : set of supply1;
end entity;

type supply1 is entity
     sno    : integer;
     pno    : integer;
end entity;
```

Local schema 2:

| supply2 | |
|---|---|
| sno | pno |

```
type supply2 is entity
     sno    : integer;
     pno    : integer;
end entity;
```

Figure 14—Two local schemata

Suppose a global schema with two entity types, "supplier" and "parts," is to be designed from two local schemata shown in Figure 14. The global schema must capture all the information contained in both schemata. Notice that in the second schema, "supplier" and "parts" entities do not exist, but their existence is implied by the presence of supplier numbers and part numbers: "sno" and "pno." To capture this information, virtual "supplier" and "parts" entities corresponding to those "sno" and "pno" must be created in the global schema. A definition of the global schema is shown in Figure 15. Notice that in the definition primary keys "supplier.no" and "parts.no" are used to map the new entities to existing entities in the first schema and the implied entities in the second schema.

*Data Incompatibility*

Several sources of data incompatibility are discussed in this section. The objective of the discussion is to show how the proposed architecture allows us to incorporate our present understanding of incompatible data into Multibase. The details of solutions to the problem are to be fully investigated later in the project.

Some sources of data imprecision are:

1. *Scale difference.* For example, in one database four values (cold, cool, warm, hot) are used to classify climates

```
type supplier is entity
     sno       : integer;
     supplying : set of parts;
  end entity;

type parts is entity
      name: string(15);
      no  : integer;
  end entity;

for each x in (sno(supplier1) union sno(supply2))
 loop
 create supplier (sno => x);
 end loop;

for each y in (pno(parts1) union pno(supply2))
 loop
 create parts (pno => y);
  end loop;

for each s in supplier loop
supplying(s) :+ (p in parts where (for some y1 in supply1:
    sno(s) = sno(y1) and pno(p)= pno(y1)) or
   (for some y2 in supply2 :
    sno(s) = sno(y2) and pno(p) = pno(y2)));
end loop;
```

Figure 15—A global schema

of cities, while in another database the average temperatures in Fahrenheit may be recorded.

2. *Level of Abstraction.* For example, in one database "labor cost" and "material cost" may be recorded separately, while in another they are combined into "total cost." Another example is recording an employee's "average salary" instead of his or her "salary history" for the previous five years.
3. *Inconsistency Among Copies of the Same Information.* Certain information about an entity may appear in several databases, and the values may be different due to timing, errors, obsolescence, etc.

There are many other sources of data incompatibility. Data incompatibility must be resolved if different databases are to be integrated. The architecture of schema integration developed previously can be extended to handle the problem.

Let E1 and E2 be two entity types, and f1 and f2 be functions defined on E1 and E2 respectively. If E1 and E2 have been merged into an entity type E, then f1 and f2 can be merged into the function f defined on E as follows,

$$f(e) = \begin{cases} T1(f1(e)) & \text{if e in E1-(E1 intersect E2)} \\ T2(f2(e)) & \text{if e in E2-(E1 intersect E2)} \\ g(f1(e),f2(e)) & \text{if e in (E1 intersect E2)} \end{cases}$$

The transformations T1 and T2 are typically used to map the ranges of f1 and f2 into a common range as discussed in the section "Merging Entity Types and Functions." On the other hand, the function g is used to reconcile any inconsistencies between the values of f1 and f2 over the same entity. Typically, g will involve accessing data described in the integration schema.

For example, in Figure 16, the entity types E4 and E5 are merged into the entity type E6 by using functions IS2 and IS3 of the integration database. In the figure, the data values of the entities and functions are shown in tabular form. In this example, T1 and T2 transform the climate of cities from two different scales, (cold,cool,warm,hot) and Fahrenheit, into a unified scale (temperature range, probability) by combining E4 with IS2 and E5 with IS3. The function g could return all the (temperature range, probability) pairs from the two databases without any further processing, as is shown in Figure 16.

Alternatively, g could use some statistical technique to process sets of (Temp range, probability) pairs, and return a simpler but descriptive summary of those pairs. For example, the function g could return the average value and the standard deviation of the distribution represented by these pairs; it can make statistical estimation and return a confidence interval; or it can do time series analysis and return information about the spectral function.

The above examples are merely illustrative of potential data integration problems and their solutions. More complete approaches to the problem will be fully investigated later in the project.

## 4. RUN-TIME QUERY PROCESSING SUBSYSTEM

### *Overall Architecture*

Now we will show how the schema mappings developed during schema integration are utilized to drive query processing over the global schema. As we explained in Section 2, the run-time subsystem consists of a query translator and a query processor. Here we will expand these two components in further detail.

A "Global Database Manager" (GDM) is that part of the Multibase System which consists of the query translator, and the query processor. A query over the global schema is normally sent to the nearest site that has a Global Database Manager (GDM). There may be one or more GDMs in a Multibase system. A GDM stores a copy of global schema,

E4 (of LS1)

| city1 | climate |
|---|---|
| Boston | cold |
| Norfolk | cool |
| Dallas | warm |
| Miami | hot |
| ... | ... |

IS2 (of integration database)

| climate | range of temp | probability |
|---|---|---|
| cold | 0 - 20 F | 20% |
| cold | 20 - 40 F | 40% |
| cold | 40 - 60 F | 25% |
| cold | 60 - 80 F | 10% |
| cold | 80 - 100F | 5% |
| cool | 0 - 20 F | 10% |
| cool | 20 - 40 F | 20% |
| ... | ... | ... |

E5 (of LS2)

| city2 | mean temp |
|---|---|
| Denver | 52 F |
| Chicago | 54 F |
| Los Ang | 75 F |
| ... | ... |

IS3 (of integration database)

| mean temp | range of temp | probability |
|---|---|---|
| 52 F | 0 - 20 F | 20% |
| 52 F | 20 - 40 F | 35% |
| 52 F | 40 - 60 F | 30% |
| ... | ... | ... |

E6 (of global schema)

| city | temp range | probability |
|---|---|---|
| Boston | 0 - 20 F | 20% |
| Boston | 20 - 40 F | 40% |
| ... | ... | ... |

Figure 16—Example of data incompatibility

local schemata, integration schema, and the mapping definitions among them. It uses this information to parse, translate, and decompose queries over the global schema into local queries over local schemata, and coordinates execution of the local queries. The structure of a GDM and its interface with local DBMSs is shown in Figure 17.

A query expressed in DAPLEX over the global schema is first parsed by the parser and a parse tree is generated. Components of the parse tree, which are entities and functions of the global schema, are then replaced by their corresponding definitions, which are expressed in terms of the local schemata LSs. The result is a parse tree consisting of entities and functions of the local schemata. The parser is part of the query translator.

The parse tree is then simplified to eliminate the inefficient boolean components. For example, the boolean expression "(a > 5)or(a < 20)" is reduced to "true," and "(a > 5)and (a < 2)" is reduced to "false." The query simplifier is also part of the query translator.

The parse tree is then decomposed by the decomposer into subtrees. Each subtree represents a local query referencing only entities and functions of a single local schema.

The "ACCESS PLANNER" transforms the local queries into "data movement" and "local processing" steps. Depending on the memory size and processing power of each individual site, and the capacity of the communication channels, the "ACCESS PLANNER" may move data and distribute the computing load among several sites, or it may move data to a central site which has large memory and computing power and do most of the processing there. In doing this planning, the "ACCESS PLANNER" tries to produce steps which minimize the cost of processing the query. The meaning of "cost" depends on the individual systems being integrated. It may mean the amount of data moved between sites, or the amount of processing time.

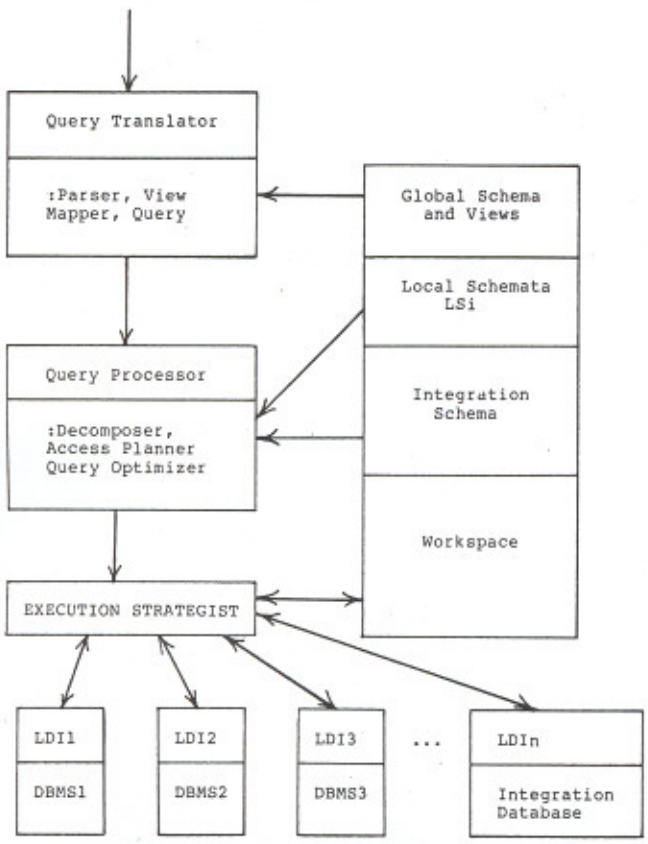



Figure 17—Run time query processing subsystem

The execution of the access plan is coordinated by the "EXECUTION STRATEGIST." It sequences the steps of the access plan and it makes sure that the data needed by a step are there before the step is initiated.

The "EXECUTION STRATEGIST" communicates with local DBMSs through the Local Database Interface (LDI). The LDIs receive "data move" and "local processing" steps from the "EXECUTION STRATEGIST," translate these steps into programs in the local query language or Data Manipulation Language (DML), or call local routines to process these steps, and translate the results of these steps into the format expected by the "EXECUTION STRATEGIST." The LDI may reside in a GDM if the local site does not have enough memory or cpu power; otherwise it resides with the individual local DBMS at the local site.

The query processor to be described in this section is oriented towards the initial breadboard system. It is designed to handle restricted versions of the user interface language and view mapping language with reasonable efficiency. Subsequent research is needed to extend the query processor to efficiently handle the unrestricted languages.

Within the "Query Processor," the database is modelled as a collection of *entity types* and *links*. A link L from entity type R to entity type S is a function from entities of S to entities of R; S is called the *owner* entity type and R is called the *member* entity type relative to L. We assume that if L links R to S, then L, R, and S are all stored at the same site. We also assume that there is a database schema describing the entity types and links of the database.

We will sketch the Multibase query processing strategy in three steps. First, we define the set of queries that can be posed. Second, we define the set of basic operations that Multibase is capable of executing. Third, we describe how to translate a query into a sequence of basic operations that solve the query. Finally, we describe how to translate a local query posed over a CODASYL local host schema into a program in a low level Data Manipulation Language.

### *Queries*

A *query* consists of a target list and a qualification. A *target list* consists of a set of *function terms* of the form A(R) where R is an entity type and A is a non-link function of R. A *qualification* is a conjunction of selection clauses, join clauses, and link clauses. A *selection clause* is a formula of the form (A(R) op k) where A(R) is a function term, op is one of $\{=, \leq, <, >, \geq, \neq\}$ and k is a constant. A *join clause* is a formula of the form (A(R) = B(S)) where A(R) and B(S) are function terms. A *link clause* is a formula of the form (L(R) = S) where L is a link from R to S.

Let r and s be entities in R and S respectively. We say that

r *satisfies the selection clause* (A(R) op k) if the A-value of r is op-related to k (i.e., (A(r) op k)). We say that r and s *satisfy the join clause* (A(R) = B(S)) if the A-value of r equals the B-value of s (i.e., A(r) = B(s)). And we say that r and s *satisfy the link clause* L(R) = S if L connects r and s (i.e., L(r) = s).

Let R1, . . . , Rn be the entity types referenced by qualification q, and let r1, . . . ,rn be entities in R1, . . . ,Rn respectively. We say that r1, . . . ,rn *satisfy the qualification* q if r1, . . . ,rn satisfy all of the clauses of q.

Let Q be a query consisting of target list T = ((Aj1(Ri1), . . . ,Ajm(Rim)) and qualification q. Let R1, . . . ,Rn be the entity types referenced in T and q. The *answer* to Q is the set of all tuples of the form ((Aj1(ri1), . . . ,Ajm(rim))) such that r1, . . . ,rn are in R1, . . . ,Rn (respectively) and r1, . . . ,rn satisfy q. Given a database R1, . . . ,Rn and a query Q, our goal is to compute the answer to Q efficiently.

The subset of DAPLEX that we have just described makes the following simplifications:

1. Set expressions in range predicates and qualifications have been "flattened out," and quantifiers eliminated. This allows us to utilize existing view algorithms for relational databases. Further research will be devoted to handling the novel aspects of view processing in the DAPLEX functional model.
2. The type-subtype hierarchy is not explicitly handled. This hierarchy will be useful in the schema integration step. However, the mechanics of interpreting queries against the hierarchy require further research.

A query graph QG(N,E) is an undirected labelled graph that represents a query Q. The nodes, N, of QG are the entity types referenced in Q. Each node is labelled by the entity type name of the node, the non-link functions of the entity type that appear in the target list, and the selection clauses of Q's qualification that reference the entity type. The edge set E of QG contains one edge (R,S) for each join clause or link clause that references R and S. Each edge is labelled by its corresponding clause(s).

A query is called *natural* if (a) join clauses are of the form (A(R) = A(S)), that is, the functions referenced in both terms of a join clause have the same name; and (b) if A is a non-link function of two entity types R and S, then A(R) and A(S) are "connected" by a sequence of join clauses. There is a simple and efficient algorithm that, given a database description and a query Q, renames the functions of the entity types where necessary to produce an equivalent natural query Q′; Q and Q′ are equivalent in the sense that they produce the same answer for any database state (up to the renaming of fields). We will therefore assume, without the loss of generality, that our queries are natural. Given that we deal only with natural queries, the edge labels corresponding to join clauses are unnecessary. Also target lists need only contain function names, instead of function terms.

Given a join clause (A(R) = A(S)) and a selection clause (A(R) op k), we can deduce that (S(A) op k). We assume that the qualification of each query is augmented by all clauses that can be deduced in this way. A simple and efficient transitive closure algorithm is sufficient for performing such deductions.

*Basic operations*

There are three types of sites in the breadboard Multibase: File, CODASYL, and GDM. Each type of site is capable of executing a different set of basic operations. This section describes these basic operations.

1. *File Select.* If record type R is stored at a File site S, then the only operation that can be applied to R at S is a selection of the form

   R[(A1 = k1) and (A2 = k2) and . . . and (An = kn)].

   The result of the selection is a record type consisting of the set of all records r in R such that r[Ai] = ki for i = 1, . . . ,n; this result is always transmitted to the GDM.
2. *File Semijoin.* In principle, File select can be generalized into File semijoin by performing selections iteratively. Let R be a File file and S a GDM file, and suppose A1, . . . ,An are fields of R and S. Then the semijoin of R by S on A1, . . . ,An, denoted R[A1, . . . ,An]S, equals

   {r in R | (there exist s in S)
   (r.A1 = s.A1 . . . r.An = s.An)}.

   This can be computed by the following program.

   ```
   Result: = 0;
   for each s in S
   loop
    k1: = s.A1, . . . ; kn: = s.An;
    Result: = Result ∪ R[(A1 = k1) . . .
          (An = kn)];
   end loop;
   ```

   In practice, this operation may place an unacceptable load on the File system and hence may not be usable.
3. *CODASYL tree queries.* The basic operation that can be performed at a CODASYL site S is to solve a natural tree query (defined below), returning the result to the GDM. A *natural tree query* Q at site S has two properties: (1) All record types referenced in Q must be stored at S. (2) Let Q′ be Q minus its join clauses (i.e., all clauses of Q′ are selections or links), and let QG′ be the query graph of Q′; then QG′ must be a tree.

   To solve a tree query Q using CODASYL DML, one essentially expands the cartesian product of the record types referenced by Q and evaluates the qualification on each element of the cartesian product. We describe how this cartesian product can be systematically generated in the section "Processing CODASYL Tree Queries."
4. *CODASYL Tree Semijoins.* The preceding operation can be generalized into a semijoin-like operation. Let Q be a CODASYL tree query and S a GDM record type, and suppose A1, . . . ,An are fields of S and fields of record types of Q. Let Q′ have the same qualification as Q, and the target list augmented by A1, . . . ,An. Finally, let R′ be the result of Q′. The semijoin of Q by S on A1, . . . ,An, denoted Q < A1, . . . ,An], equals

   {r′ in R′ | (there exist s in S)
   (r′.A2 = s.A2) . . . (r′.An = s.An)}.

This can be computed as follows. Suppose A1, . . . ,An are fields of R1, . . . ,Rn respectively where R1, . . . ,Rn are record types of Q. (R1, . . . ,Rn need not be distinct.) Augment the qualification of Q′ by adding the clauses (R1.A1 = k1) . . . (Rn.An = kn). And execute the following program.

```
Result: = 0;
for each s in S loop
  k1: = s.A1; . . . ; kn: = s.An;
  Result: = Result ∪ Q′;
end loop;
```

5. *GDM Queries.* The GDM can process any natural query Q provided (1) all entity types referenced in Q are stored at the GDM, and (2) Q contains no link clauses. Suppose Q references entity types R1, . . . ,Rn. Q is processed by constructing a request to the local DBMS (the Datacomputer for the initial breadboard system) of the form:

```
for each r1 in R1 where (selection clauses on R1)
for each r2 in R2 where (selection clauses on R2)
                  and (join clauses on R1 and R2)
        .
        .
        .
for each rn in Rn where (selection clauses onRn)
                  and (join clauses on R1 and Rn)
                  and (join clauses on R2 and Rn)
                  .
                  .
                  .
              and (join clauses on Rn-1 and Rn).
          print (target list).
```

It is important that the "for" statements be in a "reasonable" order for performance reasons. Optimization techniques developed by Wong for the SDD-1 DM[3] are directly applicable.

*Query Decomposition*

To solve a query Q, we must decompose it into a sequence of basic operations. Our basic strategy is to find subqueries of Q that can be entirely solved at File and CODASYL sites, move the results of these subqueries to the GDM, and solve the remainder of the query at the GDM.

To follow this strategy, we must isolate File and CODASYL subqueries of Q. File subqueries are easy to find. We simply find entity types in Q that are stored at File sites. For each such entity type R, we produce a subquery consisting of the selection clauses on R.

Let QG be the query graph of Q. To find CODASYL subqueries, we begin by deleting from QG all entity types not stored at a CODASYL site and all join clauses. Each connected component of the resulting graph includes entity types and links that are stored at the same site, because no link can connect two entity types stored at different sites (c.f., the section on "Overall Architecture"). If a connected component is a tree, then it corresponds to a tree query and can be solved by the CODASYL site. If it has a cycle, then it must be further decomposed into two or more tree queries. (In the breadboard version of Multibase, we will only handle queries whose CODASYL subqueries are tree queries; if some CODASYL subquery is cyclic, the query cannot be processed.)

Having extracted the File and CODASYL subqueries, we must now choose an order for these subqueries to be executed. As a first-cut solution, we propose to solve all File and CODASYL subqueries before processing the results of any of these subqueries at the GDM. This strategy will be an especially poor performer if a File or CODASYL subquery has no selection clauses. For such cases, we recommend use of File and CODASYL semijoin operations, so that the results of some subqueries can be used to reduce the cost of other subqueries. However, this tactic brings us into the realm of new query optimization algorithms and will require further research.

*Processing CODASYL Tree Queries*

Let Q be a CODASYL tree query and QG its tree. The following algorithm compiles Q into a program that solves Q. The program contains statements of the form:

1. *for* r *in* set(s) *loop . . . end loop* ; where S owns R via set ;
2. r: = set_inv(s) ; where R owns S via set. Note that set-inv is the inverse function of set and is always a function.

**Algorithm**

1. Do a pre-order traversal of QG. The result is a list of the nodes of QG. Call this list P.
2. Let R and S be nodes of QG; with R the parent of S.
   *Cases*
   R is the root of QG; replace "R" by "*for* r *in* R *loop*" in P.
   R owns S: replace "S" by "*for* s *in* set(r)" in P.
   S owns R: replace "S" by "s: = set_inv(r)" in P.
3. Push loop independent assignments up as high as possible.
4. Add an "output (target list)" statement, add selections, and joins as high as possible, tack on enough *ends* to balance the *fors*.

As an example let QG be the query graph of Figure 18.

1. Preorder traversal: R,S,T,U,V.
2. *for* r *in* R *loop*
   *for* s *in* L1(r) *loop*
   t: = L2_inv(r)
   *for* u *in* L3(t) *loop*
   v: = L4_inv(t)
3. Push up T and V; add an output statement; add *ends* to balance the *fors*.
   *for* r *in* R *loop*
   t: = L2_inv(r);
   v: = L4_inv(r)
   *for* s *in* L1(r) *loop*

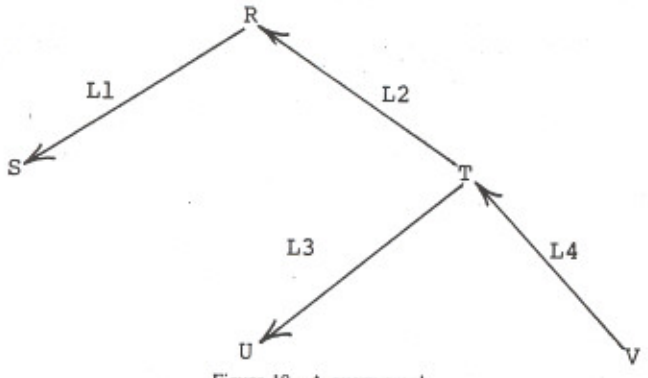



Figure 18—A query graph

```
for u in L3(t) loop
   output (target list);
end loop;
end loop;
end loop;
```

## 5. SUMMARY

This report describes the architecture of the Multibase system. Details of the components of the architecture to be implemented in the initial breadboard version are also described. Although additional research is required to fill in the details of optimization and incompatible data handling, the architecture already contains several innovative ideas in integrating distributed heterogeneous databases. These include the following:

1. the idea of using an integration database to resolve data incompatibility;
2. the idea of using a mapping language to uniformly define the global schema in terms of the local schemata and the integration schema; and
3. the idea of using query modification and query graph decomposition to transform a global query into local queries and queries over the integration database.

## REFERENCES

1. Shipman, D., "The Functional Data Model and the Data Language DAPLEX", *SIGMOD 79*, Boston, MA, 1979.
2. Stonebraker, M.R.: "Implementation of Integrity Constraints and Views by Query Modifications." *Proc. ACM-SIGMOD Conf.*, San Jose, CA, 1975, pp. 65-78.
3. Wong, E., "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases," *1977 Berkeley Workshop on Distributed Data Management and Computer Networks*, Univ. of CA, Berkeley, CA, May 1977.