NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

The Design and Implementation of a Heterogeneous Distributed Database Management System Prototype

Richard Pollock

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

October 1988

ISBN 0-315-49111-6

ABSTRACT

The Design and Implementation of a
Heterogeneous Distributed Database Management System
Prototype

Richard Pollock

A heterogeneous distributed database management system (HDDBMS) presents an integrated view of multiple pre-existing databases in order to provide the advantages of database consolidation while avoiding, as far as possible, violations of local autonomy.

An overall design for an HDDBMS where the new software acts as a front-end to multiple local DBMSs has been developed and a prototype has been implemented. The prototype involves two different commercial micro-computer based DBMSs residing on separate machines, and services queries on an integrated view of semantically-related databases that exhibit a range of schema and data conflicts.

An integrated schema is derived from relational versions of the pre-existing database schemata using a global query and mapping language (GQML). The global query processing algorithm places no restriction on data distribution or replication.

In addition to demonstrating the basic features of the design, the prototype may act as a test-bed for possible enhancements and extensions described in the thesis.

ACKNOWLEDGEMENTS

I gratefully acknowledge the guidance of my thesis supervisor, Dr. B.C. Desai, throughout the preparation of this thesis, from providing a stimulating and challenging problem to critiquing the final drafts, and engaging in many essential discussions in between. Dr. Desai arranged convenient access to every required resource as well as the financial assistance which supported me during my graduate studies. I also deeply appreciate Dr. Desai's advice on other aspects of my graduate studies, his constant availability, and his friendship.

Pat Kierans, of the Engineering and Computer Science Office at the Loyola Campus, is at the centre of the most friendly and cheerful place in which I have worked, and was a source of constant encouragement and practical assistance.

I also acknowledge the comradeship and moral support of Deb Chatterjee, Marc Drolet, and Pam Fox.

Norman Hawkins C.A. gave me generous access to his computer, which proved to be a great convenience on many occasions.

Finally, to my family, and especially my mother, Diane, I owe the greatest debt for all the understanding and help during my years of study.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 CHARACTERIZATION OF HETEROGENEOUS DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

In the design of a distributed database management system (DDBMS) we seek to provide the user with an external schema that represents data which is, in fact, distributed among multiple participating local databases. These databases may be resident on separate machines, and each has its own local schema. The user should be able to submit queries and updates on the external schema as though the DDBMS is a conventional, database management system (DBMS) managing a single physical database. It is the responsibility of the DDBMS to do the following:

a) convert a user's query or update on the external schema to a plan of one or more queries or updates on local schemata and the necessary data transfers (in the case of queries)

b) manage the execution of these operations

c) for queries, combine the results obtained from the separate local databases into a result which appears to have been obtained from a single database that conforms to the external schema.

The external schema as described here is often referred to as a "global schema" in the literature. However, we will continue to use the term "external schema" in order to

conform to the terminology developed in Chapter 2.

In [DEEN 87] a distinction is made between 'closed' and 'open' DDBMSs. In a closed DDBMS, the entire system is designed in a top-down fashion, with the local schemata designed in accordance to the requirements of a predetermined external schema. Typically, the same DBMS software is installed at each site, so each local schema is expressed using a uniform data model, which would also be the data model used to express the external schema. In contrast, an open DDBMS is designed in order to integrate existing databases managed by existing software.

The decision to implement an open DDBMS is made in order to satisfy a requirement to integrate semantically related data from multiple databases, while avoiding the upheaval that may be caused by local schema, software, or hardware changes. This situation may arise, for example, as separate organizations with their own databases are merged and it becomes necessary to answer questions related to the new organization as a whole. Another example is when workers within a single organization discover that data from multiple departments, each with its own independently developed database, is required to deal with new problems.

Regardless of the occasional requirement for integrated data, the separate departments or organizations may continue to be the most heavy users of their own databases, and have substantial investments in training and application

programs. In this case, the maintenance of the status quo with regards to data distribution, local schemata, software, and hardware is highly desirable.

The essential differences between the design of a closed DDBMS and an open DDBMS may be illustrated as follows.

Consider a closed DDBMS which, for the sake of discussion, uses a relational data model. The external schema would describe a set of virtual external relations, each of which could be divided into vertical fragments (columnwise subdivision), or horizontal fragments (rowwise subdivision) or a combination of both. The fragmentation would correspond to the allocation of data among the participating local databases. Typically, the horizontal fragments would be disjoint and the vertical fragments would only share those attributes necessary for reconstructing the external relations with join operations. Let us consider all such fragments as being disjoint, even though there may be common attributes among vertical fragments (but no more than necessary). Alternatively, joins may be done on tuple identifiers which are invisible to users, so the vertical fragments would not have common attributes. Fragments may be replicated in order to increase the efficiency and reliability of the system. In such a system, any external relation could be reconstructed by selecting copies of all its constituent fragments and combining them with join and

union operations. The processing of queries on the external schema would be based on this capability, combined with certain optimizing procedures.

In comparison, consider a open DDBMS, again using a relational data model. It cannot be assumed that data from the pre-existing local databases will correspond to disjoint fragments of conceivable external relations, nor can it be assumed that the data in the local databases will be mutually consistent. In a closed DDBMS, processing measures can be included in the design to ensure that data inconsistencies will never occur. Furthermore, in a open DDBMS design no assumptions can be made about the existing DBMS software (or operating systems and machines, for that matter).

The distinguishing issues involved in open DDBMS design may be classified as homogenization and database integration. In [DEEN 87] homogenization is described as being concerned with differences in the DBMS software which manage the local databases (a local "database" which is to be included in open DDBMS may be simply a set of files with indexed fields which are directly manipulated by a set of application programs; however, to simplify the discussion the terms "database" and "DBMS" will be used even in this context). The most important difference would be in the data models used. However, even DBMSs that use the same data model may use different data manipulation languages (DMLs)

and different file formats. The usual approach to DBMS homogenization is to implement a mapping between the local data model, DML, and file format and a global data model, DML and file format. In addition to DBMS differences, homogenization may also involve bridging operating system and machine differences.

Database integration is concerned with data and schema conflicts among local databases which exist even when local schema are expressed in the same data model. These will be referred to as 'database conflicts'. Examples of such conflicts are similar entities being described with different sets of properties, and the same property of the same entity being stored as different values in separate local databases. Database conflicts will be discussed in detail in Section 2.2.

The most important distinction between a closed and an open DDBMS is the likely requirement for database integration in the design of the latter. In the top-down design of a DDBMS a decision may be made to use existing software and hardware resources and, thus, deal with homogenization [CERI 84], but it is inconceivable that schema and data conflicts would intentionally be included in the design of local schemata. In this case, it may be argued that the DDBMS is partially 'open' and that, in fact, the concepts of 'open' and 'closed' DDBMS express two extremes of a continuum. However, the control over local schemata

afforded by top-down design does create a definite practical division between the two concepts.

In the literature the term "heterogeneous distributed DBMS" is most often used to refer to the type of DDBMS described here as "open". This term may be somewhat misleading since "heterogeneous" may be interpreted as referring exclusively to the heterogeneity of local DBMSs. However, we will assume that "heterogeneous" also refers to the possible presence of database conflicts among pre-existing local databases so that "closed DDBMS" and "heterogeneous DDBMS" (HDDBMS) may be used interchangeably.

1.2 THESIS AIM AND OUTLINE

The aim of this thesis is to propose an overall HDDBMS design and implement a prototype based on this design.

Chapter 2 establishes a terminology used in the rest of the thesis, discusses issues in HDDBMS design and how existing designs respond to them, catalogues database conflict types, and describes the general approach taken in our design.

Chapter 3 describes the global query and mapping language incorporated in our HDDBMS design. This language is based on the relational algebra and is the basic tool of database integration and conflict resolution.

Chapter 4 discusses the application of the global query

and mapping language described in Chapter 3 to resolving the database conflicts described in Chapter 2.

Chapter 5 describes the query processing algorithm incorporated in our HDDBMS design. This is the algorithm used to process queries submitted in the global query and mapping language on the integrated database schema.

Chapter 6 describes our prototype implementation based on the global query and mapping language described in Chapter 3 and the query processing algorithm described in Chapter 5. The prototype acts as a front-end to two different DBMSs which are situated on two separate microcomputers connected with a serial communication link.

Chapter 7 concludes the thesis and summarizes specific problems in HDDBMS design which were not fully addressed herein, and which are proposed as topics for further work.

CHAPTER 2 BACKGROUND

This Chapter discusses issues in HDDBMS design and surveys existing responses to them. This provides a background for the discussion of our prototype design in Chapters 3 and 5, and proposals for further work in Chapter 7. A survey of database conflicts is also presented. This provides a background for the discussion of conflict resolution techniques in Chapter 4.

Specific designs and/or prototypes which address HDDBMS issues to varying degrees and which are referred to in this Chapter are Multibase [LAND 82, SMIT 81], Mermaid [TEMP 83, TEMP 87, YU 87], PRECI* [DEEN 85a, DEEN 85b], ADDS [BREI 86], UCLA HD-DBMS [CARD 80, CARD 87], and SIRIUS-DELTA [FERR 84].

Since the terms used in the literature are often inconsistent, the terminology established in this Chapter will be used in the rest of the thesis.

The discussion of design issues is organized into Sections on schema architecture, query processing, update processing, and global data models. Existing responses are drawn from literature on specific HDDBMS prototype designs as well as from literature focussing more closely on specific issues.

This Chapter concludes with a discussion of the direction taken in our prototype design.

2.1 SCHEMA ARCHITECTURE

Both distributed and centralized DBMSs are usually designed with some recognition of a distinct schema hierarchy and facilities to support the mapping between schemata on different levels. The definitions of such schema hierarchies have been referred to as schema architectures [ELMA 81]. A well known example of a schema architecture is the ANSI/SPARC proposal for centralize databases which specifies three schema levels: internal (lowest), conceptual, and external (highest) [TSIC 77]. In general, the highest level of a schema architecture provides a user-oriented description of the database, while lower levels provide increasingly storage-oriented descriptions.

2.2.1 Contributions of a Schema Architecture to HDDBMS Design

The contributions of an explicit schema architecture to HDDBMS design include those that apply to closed DDBMS and centralized DBMS design and are as follows:

1) transparency

The top schema level is designed so that queries on it may be stated simply, with minimum concern for details about

the organization and functioning of the system. These details are embodied in the lower schema levels and the mappings between them.

2) data independence

A change to a given schema level may be made without affecting the next higher schema level as long as the mapping between the two can be modified to absorb these changes. This imparts a degree of data independence to a given schema level with respect to lower schema levels. This is especially important in distributed systems where coordination of changes at different sites may be awkward due to the physical distance between them and a desire to preserve local autonomy.

3) view independence

At specified schema levels, mappings to more than one schema at the adjacent higher schema level may be permitted. This would support the development of multiple independent views of the database which are tailored to different users. Also, the database designer or administrator can vary the data which is visible to a group of users by changing the subset of the schema at a given level which is included in the mapping to a higher level.

4) categorization of mapping specifications

Corresponding to the mapping between each pair of adjacent schemata there must be a set of specifications. These might be manually input or possibly generated with a computer aid. The use of distinct schema levels allows the generation and storage of specifications to be divided into distinct and, in the case of distributed systems, possibly location-specific components.

5) database integration

At specified schema levels, mappings to more than one schema at the adjacent lower schema level may be permitted. This would support the integration of separate databases in a distributed system.

2.1.2 An HDDBMS Reference Schema Architecture

A schema architecture for closed (top-down designed) DDBMSs is described in [CERI 84] which has the following schema levels (from lowest to highest): local database, local mapping, allocation, fragmentation, and global. A local database schema may actually correspond to the highest level of a centralized database schema architecture, but this is de-emphasized in order to focus on the distributed database issues. The types of transparency supported by this architecture are fragmentation, location, replication, and

local mapping (a relational global data model is assumed). This frees a user from having to know how a global relation is split up into different vertical and horizontal fragments, to which sites the fragments are allocated, which fragments are replicated, and the data models used by specific local DBMSs.

Note that in this schema architecture, only the local and global database schema levels actually describe database structure in terms of a data model. The other levels correspond to different kinds of information required to map between the global schema and the local database schemata, thus it is questionable whether these other levels should be called schema levels ('mapping levels' may be a more appropriate term).

The use of a global data model and associated global DML means that a DML translator is required for translating from the global DML to each local DML rather than from each local DML to each other local DML. In other words, with n local DBMSs each with a different DML, n two-way translators need to be implemented when there is a global data model, as compared with n*(n-1) translators when direct local-to-local DML mapping is done. For more than two different local DBMSs, the global data model approach requires fewer translators. Also, the use of a global data model simplifies query processing by clearly separating DML translation and global strategy formulation. Note that the DML translators

would refer to a mapping between local and global data model versions of a local database schema whenever local and global data models are not the same.

A schema architecture for a HDDBMS (designed with pre-existing databases) would have to support all the types of transparency mentioned above, plus conflict transparency. In other words, a user should not have to concern himself with the resolution of database conflicts.

The schema architectures used in existing designs for HDDBMSs all include a schema level for global data model versions of the local data model schemata, and a schema level for an integrated, conflict-free global data model view of the translated local data model schemata. In addition to simplifying the implementation of the system and query processing, as in closed DDBMSs, the use of a global data model simplifies database integration and the addition of further local databases to the system. These schema levels address the basic issues of local DBMS data model heterogeneity and database integration. However, variations abound in terminology and number of schema levels. Some designs 'compress' multiple schema levels into one or simply do not accommodate as many HDDBMS features, with respect to other designs.

We propose a reference schema architecture for HDDBMSs as shown in Figure 2.1. This schema architecture accommodates the features of all the architectures

encountered in the literature, is highly decomposed, and provides a reference terminology. In contrast to the architecture of [CERI 84], all schema levels correspond to database structure descriptions. The lines between the schema levels in Figure 2.1 symbolize mapping levels. This schema architecture does not imply any particular arrangement for the physical storage of mapping information and schema descriptions (e.g. centralized versus distributed); this would be an implementation-specific issue.

The schema levels of are schema architecture are explained below:

1) local host schema

Each site participating in the system has a local host schema that describes the local database using the data model of the local DBMS. From the local DBMS perspective the distributed system is simply another user with either its own external view of the database, or one shared with other users. There may be several local DBMS schema levels below this one, but they are of no consequence to the design of the HDDBMS. In fact, the local DBMS may itself be a DDBMS or a HDDBMS.

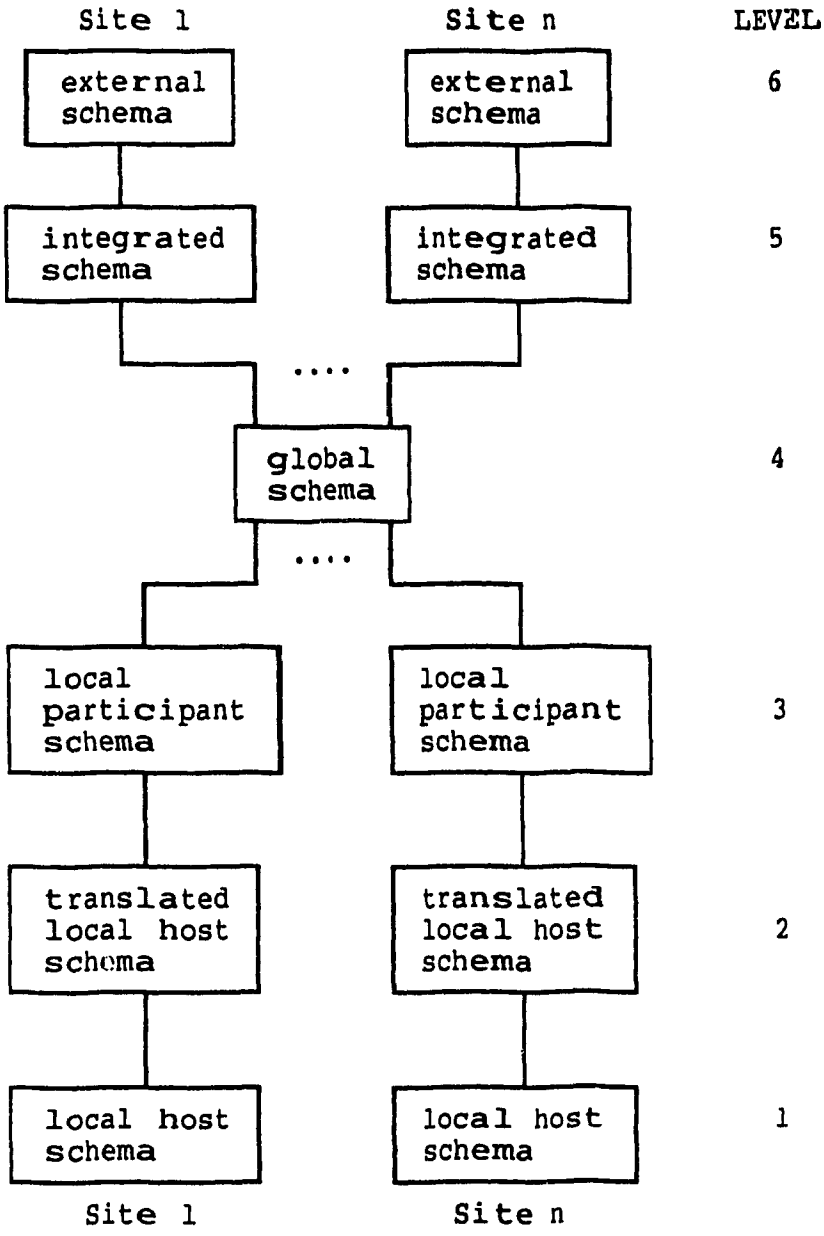


Figure 2.1 reference HDDBMS schema architecture

2) translated local host schema

This is a subset of the local host schema after translation into a global data model schema. If the local DBMS uses the global data model and the entire local host

schema is involved in the DDBMS, then the local host schema and the translated local host schema are the same. The local mapping schema of [CERI 84] corresponds to the mapping level between the local host and translated local host schema in our architecture.

3) local participant schema

This is defined by any number of mapping operations on the translated local host schema. The mapping may be performed in order to resolve a subset of the database conflicts. This could simplify higher level mappings (in which the remainder of the database conflicts would be resolved). The mapping may also be performed in order to absorb changes to the translated local host schema made after the establishment of the HDDBMS.

A local participant schema is similar to an "export schema" as described in [MCLE 80] in that it defines the view of the local database available to other sites. As such, this is a critical schema level with regards to data independence since changes to a local participant schema would entail changes to information stored at one or more other sites. This is the first level at which schemata may be designed with some knowledge of remote databases (such knowledge would be necessary in conflict resolution).

A local participant schema may be augmented by detailed

authorization controls as is the "participation schema" of PRECI* [DEEN 85]. For example, these controls may specify which properties of an object type may be updated by a request originating at a particular site.

A local participant schema may be the same as the translated local host schema, in which case no database conflict resolution mapping or authorization controls would have been specified.

4) global schema

A global schema is the collection of all local participant schemas from each site and represents the global database and the maximum amount of DDBMS data available to any one site. This is similar to the "unified global conceptual model" of the UCLA HD-DBMS [CARD 87] and the "global database schema" of PRECI* [DEEN 85].

A given site may actually have access to a only a proper subset of the global schema, thereby being aware of only part of the maximum amount of DDBMS data. This may be used as a form of authorization control.

Any global schema entity required for the response to a DDBMS query is materialized from data which is resident at one site (if the data is replicated then one site is chosen as the source of data). The mapping from the translated local host schemata to the global schema would involve

location information indicating from which site(s) data for a given global database entity is available. Thus, this mapping level corresponds to the allocation schema of [CERI 84].

5) integrated schema

Each site has an integrated schema which is defined by any number of mapping operations on the subset of the global schema accessible by that site. These mapping operations resolve any remaining database conflicts and specify integrated objects, properties and relationships.

It is possible for multiple sites to share the same integrated schema, or different sites to have different integrated schemata. The latter option would be useful if it is required to design a site's integrated schema so that it conforms as closely as possible to its translated local host schema and appears to extend rather than replace it. This property is also recognized in UCLA HD-DBMS [CARD 87] and Multibase [LAND 82] designs.

The integrated schema of a site may be the same as the subset of the global schema accessible by that site, in which case no database integration operations would have been necessary.

The integrated schema level corresponds to the global schema level of [CERI 84], and the mapping between an

integrated schema and a global schema corresponds to the fragmentation schema level of [CERI 84]. Note, however, that the fragmentation schema in a closed DDBMS would not include operations and references to metadata for conflict resolution.

6) external schema

When it is necessary for the global database at a given site to be expressed in an external data model which differs from the global data model, an external schema may be derived from the integrated schema at that site. The external schema level is similar to the "subschema level" of the Mermaid system [TEMP 87] and the "virtual layer" of the UCLA HD-DBMS [CARD 87]. If the external and the global data models are the same then the external and integrated schemata are the same.

Separate local participant and translated local host schemata, or their counterparts, with in-between mapping that resolves some database conflicts or absorbs changes to the translated local host schema is not specifically included in the existing designs of HDDBMSs encountered in the literature. It is possible that some such 'early' conflict resolution mapping capability is included in the data model translation in some systems. For example, the

translation of file system schemata to relational schemata may involve the re-clustering of fields (or attributes) to create normalized relations, and this could serve to remove differences related solely to local performance issues (see the discussion of data clustering conflicts in Section 2.2)

In PRECI* early conflict resolution operations are explicitly excluded to ensure that maximum choice for database integration and conflict resolution remains at the "global database schema" level (the counterpart of the global schema level in our architecture) [DEEN 87]. For example, suppose that one local host database stores values of a 'Weight' property as integer numbers of pounds and another stores values for the same property as integer numbers of grams. The participant schema corresponding to the latter local host database may be developed so that the units of the 'Weight' property are changed to pounds (rounded to the nearest pound) in order to resolve the conflict. In answering a HDDBMS query, the values for 'Weight' would be locally converted from grams to pounds with considerable lost of precision. The opportunity to simultaneously specify the alternate mapping from pounds to grams on the other local host database, thereby avoiding a loss of precision, would be lost since at the global schema level only pound units would be represented.

In spite of the possible danger of premature conflict resolution mapping as illustrated in the above example,

early conflict resolution mapping should not be pre-empted. In most cases it would make sense to resolve, as early as possible, schema conflicts that are related to local performance issues rather than to conceptual design logic. Also, in addition to absorbing some changes to local host databases, the capability of early conflict resolution mapping would also be useful when adding new local host databases to the HDDBMS. For example, suppose that a new local host database contains a relation which is replicated in another local host database already included in the integrated system. Furthermore, suppose that the translated local host schema of the new, 'to-be-integrated', relation has a field name conflict with the participant schema of the replica, 'already-integrated' relation. It may make sense to specify a field name mapping for the new relation so that its participant schema matches that of the replica, thereby minimizing changes to the global schema (it may still be required to add the knowledge of replicated data to the mapping information if the HDDBMS explicitly stores such information rather than acquiring it through intersite dialogue).

In Multibase design, an "integration schema" is created at the same level as the "local schemata", which are the counterpart of the local participant schemata described above [SMIT 81]. This "integration schema" describes a database which contains auxiliary data necessary for database integration and which is managed by a separate

"internal" DBMS brought in with the establishment of the HDDBMS, rather than by a pre-existing DBMS. Auxiliary data would be used to supplement a database that portrays fewer properties of a given object class than another database [LAND 82]. The "integration schema" is given special status in Multibase because it is managed by the internal DBMS. However, in our schema architecture an auxiliary database is not differentiated from a local host database since conceptually it makes no difference which DBMS manages it. In fact, in a HDDBMS of a different design, auxiliary data could be added to a pre-existing databases without any change to their independent operation (local users need not even be aware of it) and, therefore, without affecting local autonomy.

2.2 DATABASE CONFLICTS

Two databases whose schema are expressed in the same data model can conflict in two fundamental ways: with respect to the data they contain (data conflicts) and with respect to their schema (schema conflicts) [DAYA 84]. These conflicts will be discussed and further classified in this Chapter in order to provide the basis for a discussion of database integration techniques in Chapter 4.

2.2.1 Data Conflicts

Data conflicts between two databases are caused by measurement error and blunders involved in the original procurement of the data, blunders involved in their input into the separate databases, and corruption of the electronically stored data. For example, the same property-line distance can be measured simultaneously by two land survey crews using the same techniques and types of instruments, but the final quantities obtained would likely differ within expected tolerance limits due to the measurement error inherent in the techniques and instruments. Examples of possible blunders would be the incorrect recording of an intermediate or final quantity in a fieldbook, and making a typographical error while manually entering data at a terminal.

Data conflicts are fundamentally different from schema conflicts because the occurrence of measurement errors, blunders, and data corruption is essentially independent of any decisions made while designing a database. Furthermore, unless the DBMSs and supporting software are capable of preventing all such occurrences from affecting the databases (a practical impossibility), the likelihood of data conflicts cannot be discounted.

2.3.2 Schema Conflicts

Schema conflicts will be discussed with respect to record formats since virtually all commercially available DBMSs use a record-oriented data model, specifically either the hierarchical, network, or relational data model. File systems (which we consider to be eligible members of a HDDBMS), of course, are also record oriented. Accordingly, a 'schema' will be assumed to refer to a record format, including the record-type name, field names, field-types, and key specifications [KENT 82]. However, many of the conflicts discussed here also apply to more 'semantic' data models.

This section organizes concepts discussed, with considerable overlap, in [BATI 86], [BREI 86], [DAYA 84], [DEEN 87], [KENT 82], [SMIT 77], [SMIT 81], [TEOR 86]. No claim is made that the conflicts discussed below are exhaustive. They do, however, represent a comprehensive survey of the schema conflicts reported in the literature. Specific references will be given for terms, concepts, and examples (possibly modified) which are unique to a source.

Schema conflicts may be organized into the following classes:

1) name
2) value scale
3) field

4) abstraction
5) relationship
6) row/column
7) implied data
8) derived data
9) missing data
10) identifier
11) implicit/explicit truth
12) data clustering

2.2.2.1 Name Conflicts

Name conflicts pertain to record-type names, field names, and textual (character string) data, which represent actual objects, properties and relationships. They are due to the occurrence of homonyms (similar names used to identify different objects, properties, or relationships) and synonyms (different names used to identify the same objects, properties, or relationships).

2.2.2.2 Value Scale Conflicts

A value scale conflict occurs when the same property of the same class of objects is expressed in different databases using different units or different precision [DEEN 87]. For example, employee salaries may be in U.S. dollars in one database and in Canadian dollars in another. Another example is the recording of mean daily temperature of cities

in 10 degree intervals and 5 degree intervals, respectively, in separate databases. Differences in units and precision can occur simultaneously; for example, the mean daily temperature of cities may be expressed using an integral number of degrees centigrade in one database and the ordinal scale units "cold", "cool", "warm", and "hot" in another [SMIT 81].

2.2.2.3 Field Conflicts

Field conflicts consist of field-type conflicts and single-versus-multiple field conflicts.

A field-type conflict occurs when the same property of the same class of objects is represented by values in fields of different types in different databases. For example, an employee identification number may be represented by a value in a 'numeric' field in one database and by a value in an 'alphanumeric' field in another.

Another example of a field-type conflict is an employee salary represented by 'numeric' fields accommodating eight digits and nine digits, respectively, in different databases. Field-type differences may be responsible for differences in the computations permitted on the data values, and in the range and precision of data values which may be represented. Field-type differences may even lead to naming conflicts pertaining to data; for example, a field that accepts the value "color" but not "colour", and another

field where the converse is true (a general concept of field type is assumed here, where permissible value and operation restrictions included in the type definitions may encompass restrictions that are normally enforced by application programs in practical systems, if at all).

A single-versus-multiple field conflict occurs when a single field is used in one database to represent the same information that is represented by multiple fields in another database [KENT 82]. Different ways that this can happen are described below.

A single field may hold the encoded data of several fields. For example, 'Date'could be represented by separate 'Month', 'Day', 'Year' fields in one database and by a single 'Date' field in another database in which the encoding is simple concatenation (another way to regard this is that a 'Date' field is an object created from the aggregation of lower level 'Month', 'Day', and 'Year' objects - see Section 3.2.5). An example of more elaborate encoding is a personal identification number which includes digits to indicate sex and date-of-birth.

A single field may have alternate meanings represented by several fields in another database. For example, a record type for representing employees may include separate fields for 'Sales_Commission', used only for salesperson employees, and 'Overtime', which is never used for salespersons. An alternate record type for the same employees may be a

'Comm/Over' field [KENT 82].

A single field may encompass the domains of multiple fields in another database. For example, a record representing users of company cars may have separate fields for department names and employee names (so that cars may be signed out to an entire department or to an individual employee). The domains of the separate fields would be the set of all department names and the set of all employee names, respectively. An alternate record type may simply accept both kinds of names into the same field, with another field to indicate 'User_Type' [KENT 82].

2.2.2.4 Abstraction Conflicts

Abstraction conflicts occur when different databases contain semantically related information with differences in the generalization of objects and properties, the summarization of properties, set abstraction [DAYA 84], and aggregation [SMIT 77].

An example of a generalization difference is exhibited by one database using a single record type to represent all employees, while another uses several record types to represent employees according to job speciality, e.g. 'Architect', 'Engineer', or 'Planner'. Another example is the use of separate fields for 'Home_Phone#' and 'Work_Phone#' in an employee record in one database, and the use of a separate record type containing 'Phone#' and

'Emp_No' fields in another database, where no distinction is made between different kinds of phone numbers [DAYA 84].

A difference in summarization of properties (also called "statistical aggregation" [CODD 79]) is exhibited by one database associating an average departmental salary with each employee while another records the exact salary of each employee [DEEN 87].

A set abstraction difference is seen when one database represents convoys of ships (essentially, sets of ships) while another database represents individual ships. Some properties of the convoy set abstraction may be summaries of properties of individual ships in the convoy, e.g. 'Avg_Weight' to summarize the 'Weight' of member ships [DAYA 84]. Set abstraction has also been referred to as "cover aggregation" [CODD 79].

Aggregation, as explained in [SMIT 77] occurs when related objects are made into a single, higher level object. In [CODD 79] this is called "cartesian aggregation". Field types are aggregated to create a record type (in this case, the field types take on the role of 'property', but this does not disqualify them from being regarded as aggregated objects as well). Essentially, we are concerned here with differences in the distribution of field types among record types for semantic reasons; when such differences occur for performance reasons and do not correspond to different intended semantics, they are classified as data clustering

conflicts (see Section 2 ..2.12). Single-versus-multiple field differences may also be considered as aggregation differences, but these are classified under field conflicts in order to maintain a record-oriented data model perspective.

An example of an aggregation difference, in our limited sense, is the division of employee data into 'Personnel' records and 'Payroll' records in one (or, possibly, two) database, while the same data is represented by a single 'Employee' record type in another database. Admittedly, this difference may have performance and storage consequences; for example, in the first database it would be possible to store personnel data for retired or just hired employees for whom there is no payroll data, without having to store as many null values or blanks as would be the case in the second database. However, the division of data would also conform to the fact that to database users in different departments, an employee and her payroll account represents distinct objects.

2.2.2.5 Relationship Conflicts

The same objects may be portrayed with the same level of abstraction in different databases, but there may be schema conflicts due to differences in the inter-object relationships portrayed by the schemata. For example, consider the objects 'employees' and 'departments' which

have a many-to-one relationship in one database and a many-to-many relationship in another. If the databases use the relational model, the relationship may be implemented in the many-to-one case by including the department identifier attributes as foreign keys in the 'Employees' relation and, in the many-to-many case, by a 'WorksIn' relation containing both the department and the employee identifier attributes. If the databases use the CODASYL DBTG network model, then in the many-to-one case the 'Department' record type could be the owner of a set that has the 'Employee' record type as its member. In the many-to-many case, both the 'Employee' and the 'Department' record types would have to be owners of sets which would have a common 'link' member record type.

Relationships among three or more object classes may be defined in different ways. For example, in the case of three object classes, there may be a ternary relationship connecting all three, or there may be two binary relationships [TEOR 86].

2.2.2.6 Row/Column Conflicts

Row/column conflicts occur when information represented by field values in one database are represented by field names in another database. An example of this is a 'Sales' record-type which has 'Month' and 'Sales' fields in one database and separate monthly-sales fields ('Jan_Sales', 'Dec_Sales', etc.) in another database. The information

conveyed by the values of the 'Month' field in the first database is conveyed by the names of the monthly-sales fields in the second database [KENT 82].

2.2.2.7 Implied data Conflicts

Implied data conflicts occur when important data implied by the context of different databases are not accommodated by, or is omitted from, their schema.

For example, consider a record type with fields named 'Name', 'Owner', 'Rating', and 'Telephone' to describe restaurants. Suppose that two separate files using records of this type exist in Montreal and Toronto. When records of either file are displayed to users in the corresponding city, the property 'Location' would be implicit. However, if the data from both files were simply written to one file using the same scheme, location data would appear to be missing to a user who realizes that the represented restaurants may be located in either city [DEEN 87], or the data would be interpreted incorrectly by a user who still assumes that all the restaurants exist in one city.

Field names may be adequately meaningful in a particular context, but inadequate in a global context. As an example of this, consider separate databases which portray a property of what we assume to be a single employee, using a field named 'Salary' in both cases. If the two databases are regarded globally, with no knowledge of

their specific contexts, and the value in the 'Salary' field is not the same in both databases, then it is uncertain whether there is a data conflict, whether the employee works for two departments and draws a separate salary from each, or whether employee's salary has recently been changed and only one database has recorded the change due to different update schedules.

Suppose further that we are assuming that the records in question represent the same employee because an 'EmpNo' field exists, and holds the same value, in both records. If the system of employee numbers in the two databases are different we may, in fact, be looking at the salary of two different employees.

In [DAYA 84] the possibilities of the above 'employees' example are said to represent a data conflict where the separate databases are mutually inconsistent but correct. However, we do not classify this as a data conflict since the perception of mutual inconsistency is based on the interpretation of the schemata, rather than on measurement errors or blunders.

Naming conflicts are distinct from implied data conflicts in that the former may occur due to the fact that different database designers may choose a different word for the same object, property, or relationship even if they are designing a database for the same purpose and in the same operational context. In this case, knowledge of context is

usually not useful in detecting the conflicts; they can only be discerned by finding adequately descriptive explanations of what the chosen words are supposed to represent.

2.2.2.8 Derived Data Conflicts

Data which is derived from stored field values in one database might be explicitly stored in another database for performance reasons, even though they could be derived there too. This would result in differences in the record structures of the two databases.

Derivable data might be explicitly stored in order to reduce data retrieval response time by reducing the computation involved or by reducing the number of different records that might have to be accessed [KENT 82]. Updates would be complicated, but this could be an acceptable trade-off.

2.2.2.9 Missing Data Conflicts

A missing data conflict occurs when the same object class is portrayed in two databases, but one database contains data pertaining to a property of objects of that class which does not exist in the other database. Furthermore, the missing data cannot be derived directly from the existing data or the database context. A simple example are two files in separate databases each of which represents 'Employees' and each of which have the same

fields except that one file has 'HomePhone' field (and the corresponding data) and the other does not.

The reason for the difference would be that in the database where the data is missing, the corresponding property is not considered relevant, the data could not be obtained, or no one thought of including it. In the above example, in the organization that uses the database without the 'HomePhone' data, this data may be extracted from a telephone book when needed. Also, even though users may wish the data were in the database, there is not quite enough incentive to make the improvement.

If data is only portrayed differently in different databases (e.g. row/column conflict) and is not actually missing from one database, then a missing data conflict does not exist.

2.2.2.10 Identifier Conflicts

An identifier conflict occurs when the same class of objects has different unique identifier properties in different databases. For example, two different databases may identify employees by social insurance number and an arbitrary employee I.D. respectively. The two corresponding field names, say 'SIN' and 'EmpID', cannot be considered to be homonyms since they represent different properties, apart from their roles as identifiers - 'SIN' values represent federal account numbers and have uses which 'EmpID' values

do not have.

As a slightly more complex example, consider one database which identifies employees by their social security number and another which uses citizenship and social insurance/security number for the same purpose. In the first database it is assumed that all employees are Canadian and that all Canadians may be identified by their social security number. In the second database, both Canadian and U.S. employees are recorded and the possibility that a valid U.S. social security number may match a valid Canadian social insurance number is acknowledged (the field name 'SISN' is used for both numbers, out of convenience, although the semantics may be "federal account number in country of citizenship").

Identifier conflicts are special cases of missing data conflicts or implied information conflicts since the data used in identifying objects of a particular class in one database is excluded from another database which also portrays the same class of objects.

2.2.2.11 Implicit/Explicit Truth Conflicts

Normally, data is kept by recording facts which are definitely known to be true, so the truth value of the recorded facts is implicitly 'true'. However, it is also possible to record both true and false facts with an associated explicit truth indicator. For example, the

following two tables 'Tabl' and 'Tab2' contain the same information (if the closed world assumption is taken for 'Tabl'), but they exhibit an implicit/explicit truth value conflict since in 'Tabl' the truth of the facts is implicit while in 'Tab2' the truth of the facts are explicitly given by the 'Truth' attribute value (example from [KENT 82]). In this example, a 'fact' is a correspondence of 'FA' and 'FB' values in the same tuple.

Tabl (FA	FB)
A1	B1
A2	B2

Tab2 (FA	FB	TRUTH)
A1	B1	true
A2	B1	false
A1	B2	false
A2	B2	true

Note that in this example there are no field name conflicts since 'FA' and 'FB' represent the same properties in either database.

In the explicit truth approach, the intent may be to record all facts and their associated truth value, but it is obvious that this would often be impractical because of the amount of data which would have to be stored (hence the predominance of the implicit truth approach). However, if this can be done, the updates to the facts may be easily validated, i.e. if a proposed fact does not exist in the database as either 'true' or 'false' then it is not valid.

In the explicit truth approach, it may be the case that not all facts are recorded, and so the truth value of unrecorded facts cannot be determined from the database.

This is almost similar to the implicit truth approach with the database interpreted using the open world assumption where 'true' facts may indeed exist while not being recorded (all we can say is that the recorded facts are definitely true). The difference is that with the explicit truth approach some facts may be recorded as 'false' while with the implicit truth approach, no fact can be shown to be definitely 'false'.

2.2.2.12 Data Clustering Conflicts

There is no reason to believe that separate databases which represent the same facts with the same fields with the same level of abstraction will automatically be designed with the same record structures. Although normalization theory may suggest desirable record structures for avoiding storage of redundant data, null values, and loss of information, it can leave room for variations in overall design [KORT 86]. In addition, one cannot assume that different designers have the same inclination to use normalization theory, and "denormalized" designs may even be deliberately produced for performance reasons [SCHK 81]. For example, a normalized relational database could have separate 'Employee' and 'Department' relations because there are many employees in each department. However, in a real database where certain department and employee data are almost always accessed together, some department data may be included in the same records as employee data in order to

lower response time for queries. This would, of course, introduce considerable data redundancy, but this may be an acceptable trade-off for the data retrieval benefits. Another database containing the same information may be designed with different queries in mind, and the normalized design may be used. As a result, there would be differences in vertical (columnwise) clustering of the data in the two databases. Note that in the above example, denormalization was not performed to change the conceptual object classes.

Conceivably, in a given database, a large file might be split up horizontally into separate smaller files having the same record structure, representing the same object classes, and without any clear relation to the semantics of the data. This could be necessary in environments with very primitive memory management, and could result in differences in horizontal (rowwise) clustering of data in different databases.

It may be argued that data clustering conflicts are due to low level 'internal' schema differences and would not be visible in the 'conceptual' or 'external' schemata of the database. However, in a real situation we cannot assume that a given local host database will adhere to the ANSI/SPARC schema architecture, or any other proposed schema architecture for that matter. In fact, this is the case for file systems and most micro-based DBMSs.

2.3 QUERY PROCESSING

2.3.1 Global Query Processing

A HDDBMS converts a single user query on an integrated schema at one site into a set of subqueries to be distributed among one or more sites, and a set of data transfers between sites. Furthermore, the HDDBMS must coordinate the submission and execution of these subqueries and data transfers so that ultimately a response to the initial query is produced at the query site. This collection of activities will be referred to as 'global query processing'. The aspect of global query processing concerned with decomposing a query on the integrated schema (the 'global query') into a set of subqueries will be referred to as 'global query decomposition'. The mapping specified between the global and integrated schemas as part of database integration is essential to global query decomposition. It is possible for query decomposition to produce a single subquery which is, in fact, the original query and is executable at one site.

A major concern of global query processing is producing a strategy that minimizes response delay as far as is practical. This is especially important when processing ad hoc queries. This aspect of global query processing will be referred to as 'global query optimization'.

The type of links that connect HDDBMS sites is an

important factor from the optimization point of view when designing a global query processing algorithm. In the context of a long haul communications network such as ARPANET where data communication delay dominates query execution (or data processing) delay, elaborate strategies for reducing the volume of data to be transferred between sites are worthwhile. Examples are the semijoin algorithms used in SDD-1 [GOOD 81] and Mermaid [TEMP 87], and the algorithms based on estimating the costs of different join sequences, join methods, and join sites used in R* [DANI 82] and Multibase [DAYA 85]. In the context of a fast local area network, communication delay does not overwhelm data processing delay and extra operations such as semijoins that simply reduce data volume are not profitable. A global query processing algorithm designed for this context is the fragment-and-replicate algorithm used in distributed INGRES [EPST 80] and Mermaid [YU 87]. This algorithm attempts to make use of semantic information to distribute a global join as multiple locally executable joins which can be executed in parallel at different sites.

Extreme heterogeneity would include varying network types within a single integrated system. In [TEMP 87] it is reported that work has started on a combined semijoin/fragment-and-replicate algorithm to deal with this type of heterogeneity to some degree.

Global query processing might involve the complete

generation of a plan, followed by its use (static planning) or planning might be interspersed with execution steps (dynamic planning), as in the Mermaid semijoin algorithm [TEMP 87]. The advantage of dynamic planning is that estimates of intermediate result sizes upon which strategic decisions are based can be validated by comparing them to the actual intermediate results, and the plan can be altered in response to a discrepancy. In static planning, estimation errors may grow uncontrollably. However, dynamic planning usually requires greater intersite communication.

Variation in performance and capabilities of local DBMSs is a recognized factor in global query processing. Multibase uses a weighting factor for each site which represents the relative performance of the local DBMS, based on whether or not the local DBMS can perform the equivalent of joins, semijoins, and set operations using sorting and merging [DAYA 85]. Variation in capability is important when some local DBMSs cannot perform equivalents of all operations specifiable in the global query language, or when the equivalent operations are difficult to generate from global query language expressions. In the Multibase system, a "filter" is used in query decomposition to pick out operations that the current plan would send to sites that cannot support them (the plan is then modified accordingly). The Multibase system includes its own DBMS at one or more sites which can compensate limitations in local DBMSs [LAND 82]. In PRECI*, a subsidiary DBMS capable of supporting all

global query language operations is associated with each local DBMS (besides compensating for local deficiencies, the subsidiary DBMS also manages metadata related to schema levels above the local host schema level) [DEEN 85b]. The Mermaid system simply requires that a local DBMS support a minimal set of operations [TEMP 87]. This issue is further discussed in Section 3.4 in the context of global query language design.

2.3.2 External Query Processing

In the case where user queries are submitted on an external schema which is separate from the integrated schema, the HDDBMS must also transform this 'external query' into an equivalent set of global queries, control their submission to the global query processor, and transform the integrated schema response into the appropriate external form. This will be referred to as 'external query processing'. This processing requires the mapping that is specified between the integrated and external schemata (if they are not the same), as well as an external-to-global query language translator and a data format translator.

In many cases, an external query would simply be translated into a single global query, but this might not always be so. Consider the natural language query "are there any employees or vehicles at warehouse A?" where the integrated schema models employees and vehicles as separate

entities, each with a warehouse location attribute. An external query processing strategy might be to create two separate global queries corresponding to "are there any employees at warehouse A?" and "are there any vehicles at warehouse A?", submit one of them to the global query processor, and then only submit the second if the result to the first one is empty. The appropriate external response might be "Yes" or "No" while the result produced by one of the global queries would be a table representing a result relation, assuming a relational global model.

In most HDDBMS designs encountered in the literature, users are expected to submit queries in an external query language which is the same at all sites. Mermaid does include translators for SQL, QUEL, and ARIEL external query languages, but the external and integrated schemata are the same [TEMP 83]. Overall, only simple external query processing has been incorporated in HDDBMS designs, but this reflects the level of sophistication of most of the available DBMS user interfaces.

2.3.3 Local Query Processing

In all HDDBMS designs encountered in the literature, subqueries are initially expressed in a global query language and then translated into a locally processable representation, in order to minimize the number of required query language translators (as explained in Section 2.1.2)

and to simplify global query processing and the addition of new local DBMSs to the system. In most HDDBMS designs, the subqueries are also transferred between sites in the global query language.

The translation of received subqueries into a locally executable representation will be referred to as 'local query processing'. There are several aspects to this activity. Global query language subqueries sent to a remote site refer to the local participant local schema of that site, and so must be modified into a query on the local host schema of that site if the two schemata are not the same. This requires the mapping specifications between the local participant schema and the translated local host schema, and between the latter and the local host schema. In addition, the local DBMS which is meant to execute the subquery may not accommodate the global query language, in which case the HDDBMS must translate the global language query on the translated local host schema into a language that is understood by the local DBMS. This may even be necessary if the local DBMS data model is the same as the global data model. This language translation may be quite complicated. For example, it may involve the generation of a data processing program in a language such as DL/1 (used by the IMS hierarchical DBMS) from a query in a relational algebraic global query language. The generation of efficient locally executable query expressions, particularly record-at-a-time programs from queries in a set-oriented global

query language, may require a degree of intelligence. This aspect of local query processing will be referred to as 'local query optimization'.

For example, the Multibase system, which uses the functional data model as its global data model and DAPLEX as its global query language, can generate a CODASYL DML program for a limited class of global query language queries (the DBMS included as part of Multibase would handle queries outside of this class). The cost of each possible CODASYL DML program is computed, based on the number of disk page accesses involved, and the cheapest program is selected for execution [DAYA 85].

In PRECI* local query processing includes assigning operations not supported by the pre-existing local DBMS to the associated subsidiary DBMS [DEEN 85a].

The Multibase prototype described in [LAND 82] incorporates a hierarchical and a network model local DBMS in addition to its own functional data model DBMS. The SIRIUS-DELTA prototype described in [FERR 82] incorporates a relational and a network local DBMS. The Mermaid prototype described in [TEMP 87] incorporates only relational local DBMSs but these use a variety of query languages, namely SQL, QUEL, and IDL. The PRECI* prototype described in [DEEN 87] incorporates two relational PRECI/C local DBMSs. The literature on other HDDBMSs does not clearly indicate which local query languages are supported by working prototypes.

2.3.4 Data Interchange

Local DBMSs store data in files which conform to a certain structure and may contain ancillary information on that structure. For example, the data files for a simple tabular DBMS would contain a header detailing the names, order, types, lengths, and record displacements of each field of a record. In some databases, data files may also contain pointers to records in other files. Also, indexes would be stored as files holding pointers to records in data files and possibly to records in the same file.

The structure of data files and index files is traditionally very DBMS specific. This is important to HDDBMS query processing because it means that a data file or an index file from one site may not be usable by a local DBMS at another site, even if the two DBMSs conform to the same data model.

Conceivably, the data values may be extracted from a data file, converted to a stream of ASCII characters, and then transferred to another site and converted into a local data file. However, further information would be required to re-impose a structure on this data and make sense of it.

A uniform structured data interchange form (SDICF), such as the one described in [CHIA 81] may be used to support the interchange of data in a HDDBMS. The SDICF would consist of two sections, comprising the data values

themselves and a description of the data. The corresponding data file could be built directly from a SDICF file with only the information contained in that file and an understanding of the SDICF protocol. The SDICF would have to be designed with regards to the data transfer protocol in use. In many cases, this would mean that all characters in an SDCIF file be ASCII in order to avoid confusing data and the control characters recognized by the data transfer protocol [CAMP 87]. Each site in the HDDBMS would require a 'data formatting' facility to translate a SDCIF file to a local database file and vice versa (this facility might even be part of the local DBMS). The use of a SDCIF parallels the use of a global query language in that the number of data formatters is minimized with this approach when more than two different DBMSs are incorporated into the system, and the addition of new sites is simplified.

In [GLIG 84] typical SDICFs are criticized for not accommodating structured data with pointers, and consequently not allowing indexes and data files with pointers to be exchanged between sites. It is suggested that a more powerful "structured data transfer protocol" which would accommodate pointers is required, but it is also stated that very little has been done in establishing such a protocol.

The format in which queries is exchanged between sites is also an issue, even if a global query language is used.

For example, if a query is exchanged in a decoded, low level form such as an operator graph (for a global query language based on the relational algebra), processing at the receiving site may be reduced. However, unless such a representation is expressed exclusively with ASCII characters (i.e. no pointers) it would be incompatible with many data transfer protocols. Highly encoded ASCII representations appear to be mostly used in practice, for example DAPLEX queries in Multibase [LAND 82] and Distributed Intermediate Language (DIL) queries in Mermaid [TEMP 87]. Also, human readable representations have obvious convenience in an experimental system.

2.4 UPDATE PROCESSING

Let us refer to the deletion and/or addition of data to a database as updates. Users of an HDDBMS ideally should be able to perform updates on the external schema, within the constraints incorporated into that schema, so that the changes are reflected in the results to subsequent queries. The HDDBMS must accommodate these 'global updates' by making corresponding 'local updates' to one or more individual local host databases. This is an extremely difficult problem in HDDBMS design and very little has been achieved towards its solution.

Most HDDBMS designs simply do not allow global updates (in fact, most HDDBMSs are experimental and global updates

are usually outside of the project scope). Updates would be performed directly through the local DBMSs as in Multibase [LAND 82].

In [TEMP 87], it is reported that the Mermaid system was to be enhanced to support interactive updates, but only to local host database entities at one site at a time. Since higher schema levels are bypassed, this would simply provide a uniform update interface to the separate local DBMSs (which are all relational in the prototype).

In [DEEN 85] it is reported that PRECI* was to be developed to support updates on "base relations" (relations in local host schemata) only, as in the Mermaid system, but that base relations from different sites could be referenced in the same transaction, unlike Mermaid. Also, if an update is made on a replicated base relation, the system would take care of performing updates on secondary copies after updating the primary or "master" copy. The master copy is guaranteed to be up to date in PRECI*, and a user can submit a query in "Mode L" in order to guarantee that only master copies are referenced. A query submitted in "Mode A" will use either secondary or master copies, whichever would result in less delay, so it is possible that a secondary copy to which an update has not yet been propagated will be referenced.

The update problem can be divided into two subproblems: 1) Mapping updates across schema levels, and 2) distributed

transaction management. These will be discussed separately below.

2.4.1 Mapping Updates Across Schema Levels

An external or integrated schema is essentially a view derived from the local host schemata. Obstacles to deriving equivalent updates on base database entities from those on view database entities have long been recognized in relational database design (where the database entities are relations) [DAYA 78].

For an example of such an obstacle, consider two base relations with the following schemata:

EMP (EMPNM ADDR DEPTNM)
DEPT (DEPTNM LOCATION MGRNM)

Now suppose that a view is defined from these base relations as a natural join followed by a projection to obtain the following schema:

EMPMGR (EMPNM MGRNM)

Additional information would be needed to determine whether changing the 'MGRNM' value of an 'EMPMGR' tuple should be mapped to a change to the 'DEPTNM' attribute value of a tuple in the 'EMP' base relation, or a change to the 'MGRNM' attribute value of one or more tuples in the 'DEPT' base relation.

Such obstacles to update mappings may be expected to be more severe in a HDDBMS than in a centralized DBMS with a

view facility, or even a conventional DDBMS, because of the special operations that may be needed for database integration. These operations may, for example, derive virtual properties as functions of base relation properties, or (if a relational global data model is used) transpose rows and columns of attribute values. Operations for database integration are discussed in detail in Chapters 3 and 4.

The database integration mapping may be performed in such a way as to simplify update mapping, but the usefulness of the integrated schema would probably be compromised. This would not be an acceptable trade off if global queries are made much more often than global updates, as would usually be expected.

Any interschema mapping which occurs at a local site may also present update mapping obstacles, and these might not be encountered until local update processing is carried out.

At best it may only be possible or practical to map a global update to local updates in only a few cases.

Another problem related to interschema mapping and database integration is the possible conflict between update constraints imposed by the local DBMSs and the global constraints required to preserve the validity of database integration mappings. The local constraints would be a

subset of global constraints since the pre-existing databases are the starting point for database integration. This means that the possible updates through local DBMSs may be restricted due to participation in an HDDBMS, thus violating the principal of site autonomy. For example, if the integration mapping assumes that two databases represent disjoint populations of the same class of objects, then a local update which causes an object from this class to occur in both databases would invalidate the mapping. Very little attention has been given to this problem, and at the moment it appears that some violation of local update autonomy by making local constraints more restrictive to match global constraints is a necessary price to pay for participation in a HDDBMS.

2.4.2 Distributed Transaction Management

Even if global updates can be mapped to local updates (or if the HDDBMS allows location transparent transactions on base entities as in PRECI*) distributed transaction management, consisting of global concurrency control and global recovery control, is a difficult problem in a HDDBMS.

2.4.2.1 Global Concurrency Control

As stated in [GLIG 84], if two global transactions, A and B, (representing global updates) are executed concurrently and reference shared local data, the

distributed transaction manager (DTM) must make sure that the results are the same as A executed before B or B executed before A, i.e. some serial execution of the global transactions. In [GLIG 84] two alternatives for achieving this are considered:

1)Coordinate the concurrency control mechanisms of the local transaction managers (LTMs).

2) Implement the DTM global concurrency control mechanism as software existing on top of LTMs which ensures that at all sites where subtransactions A_i and B_i share data, either A_i effectively precedes B_i, or B_i effectively precedes A_i. By A_i "effectively precedes" B_i it is meant that the two subtransactions may in fact be running concurrently, as long as the LTM guarantees the serialization order of A_i before B_i.

The first alternative is considered impractical since it would be difficult to develop a general scheme for coordinating any combination of the many concurrency control algorithms. Furthermore, this alternative would require modifying the LTMs so that they recognize and respond to one another. This would require extensive reprogramming, and assumes that access to source code is possible in the first place.

The second alternative is attractive because it appears to be simpler and does not require modification of LTM

software. If subtransaction ordering cannot be specified through the local DBMS interface, then the DTM could simply submit the subtransactions in the required serialization order at that site [GLIG 86]. The DTM would have to ensure that multiple subtransactions of the same transaction, intended for the same site, are combined and actually submitted as a single transaction in order to prevent a local transaction (submitted directly to the local DBMS, thus bypassing the HDDBMS) from being inserted between them. The DTM would have to be able to identify the objects accessed by each subtransaction in order to determine whether it is necessary to specify a precedence among subtransactions at the same site. In [GLIG 86], variations of existing algorithms for concurrency control (e.g. two-phase locking, timestamps, serialization graph checking) are advocated as the basis for a DTM algorithm, as long as deadlock detection rather than deadlock avoidance is used. Deadlock avoidance is considered to require too much information from the individual sites. This information would not even exist at sites that use deadlock detection.

2.4.2.2 Global Recovery Control

For the same reasons that it is desirable to build global concurrency control mechanisms above those of the LTMs, it is desirable to build global recovery mechanisms this way too. It would be assumed that all local DBMSs have recovery mechanisms. The DTM must ensure that all

subtransactions complete or abort.

In [GLIG 84], a two phase commit protocol is advocated as the most practical global recovery approach. The DTM must be able to view each subtransaction in one of four states: 1) active (running but no updates stored in the database), 2) aborted (terminated and all updates removed from the database), 3) committed (terminated and all updates permanently stored in the database), and 4) prepared (all updates have been stored in a safe place). In the event of a system crash, a prepared subtransaction that hasn't yet been committed can be committed when the system recovers. The protocol is based on preparing all subtransactions in the first phase and committing them in the second. It is noted that most local DBMSs do not support a prepared state, so in fact, they would have to be modified to support this state. This is considered an unavoidable trade-off for a practical algorithm.

In [GLIG 86] it is noted that two-phase protocols are "blocking", meaning that if the coordinator site crashes, subtransactions remain blocked until the coordinator site recovers. This is a threat to local autonomy, since objects held by blocked subtransactions are not available to local transactions that bypass the HDDBMS. It is noted that non-blocking protocols such as the three-phase protocol exist, and that these may offer a solution to this problem.

2.5 GLOBAL DATA MODELS

The following are the basic requirements of a HDDBMS global data model:

1) It must be possible to define an equivalent global data model schema for any local host database schema, within the accepted scope of local DBMS types (typically, this would encompass hierarchical, network, and relational model based DBMSs).

2) The global data model must support an adequate database integration (including conflict resolution) capability.

3) The global data model must support a data manipulation language at least as powerful as any of those used in the local DBMSs.

Further practical considerations are as follows:

1) How easy is global query processing and update processing under the global data model?

2) How manageable is the database integration task under the global data model?

3) How straightforward is the schema and operation mapping between the global and local data models?

4) How easily are the data model entities exchanged between sites over existing communication links?

Most of the HDDBMS designs and working prototypes encountered in the literature use a relational global data model. Prominent exceptions, which will be discussed later, are the Multibase system and the UCLA HD-DBMS. The relational model offers many advantages in as an HDDBMS global data model. First, the relational algebra is easily extended with new operations to make it a mapping language with powerful conflict resolution capabilities [DEEN 87], [EREIT 86]. An integrated schema would be defined as a set of relational views derived from the global schema using the extended relational algebra operations. Likewise, a local participant schema would be a set of relational views derived from a translated local host schema. The mapping language and the global query language (for expressing queries on the integrated schema) can be one and the same. Thus a global query can be treated as an extension to the mapping, and both can be considered together as a single connected graph of operations. This simplifies query decomposition [DEEN 85].

Second, considerable work has been done on defining relational views of hierarchical and network data model schemata, and in translating relational algebraic operations on those views into the DML of the underlying data model [KAY 75], [ZAN 79], [ROSE 82]. Several commercial products have even been developed for this purpose, for example Cullinet's IDMS/R which provides a relational interface to a CODASYL IDMS database. Also, queries in common predicative

query languages such as QUEL and SQL are easily translated into relational algebra expressions [GRAY 84].

Third, relations are easily modelled as simple flat files without pointers, conforming to a standard database interchange format, for data transfer between sites.

Specific HDDBMS designs using a relational global data model for which working prototypes have been constructed include PRECI* [DEEN 85b], Mermaid [TEMP 87], ADDS [BREIT 86], and SIRIUS-DELTA [FERR 82]. All use a relational algebra based mapping language, except for Mermaid which uses a relational calculus based language. However, Mermaid has by far the most limited conflict resolution capability.

The Multibase system uses the functional data model (FDM) for its global data model and DAPLEX as both a mapping language and a global query language [SMIT 81]. In the functional data model, "entities" represent real-word objects and have "functions" representing properties of these entities or relationships among them. This is roughly equivalent to the use of tuples and attributes in the relational model, so a relational schema is easily mapped into a FDM schema.

In the FDM, functions may be multivalued, so a CODASYL network schema is translated into a FDM schema in a very straightforward manner: record types and set types are directly mapped into entity types and functions

respectively, and fields of records are also mapped into functions. Repeating fields are mapped into multivalued functions. The mapping of a CODASYL schema into a relational schema (where attributes must be atomic) is not as simple since a record type with repeating fields has to be mapped into multiple relations, and foreign key attributes have to be derived from pointer values [ROSE 82].

It must be remembered, however, that in order for local DBMSs to perform data processing, the data must be structured according to the local data model. Thus at a site with a relational local DBMS in a HDDBMS using the FDM as the global data model, the schemata for intermediate results and imported data must be translated from the FDM representation to the relational representation. This task would be as complex as generating a relational schema from a CODASYL schema.

The DAPLEX DML used as a mapping language in Multibase has a complex syntax where entity-at-a-time looping is mixed with set operations. A rich set of constructs is provided, including case statements and operators to derive new functions from existing functions. Consequently, DAPLEX is a more powerful tool for database integration than a basic relational algebraic DML (examples of the use of DAPLEX in database integration are given in [LAND 82]). However, as shown in [DEEN 87], a relational algebraic language may be extended to provide equivalent database integration

capabilities with a simpler, more concise syntax.

In the UCLA HD-DBMS design [CARD 80, CARD 87], the integrated schema is expressed using the entity relationship (E-R) model, augmented with a parallel schema in an adaptation of the Data Independent Accessing Model (DIAM) [SENK 73]. Essentially, the use of two parallel schemata is an attempt to separate the conceptual (E-R) aspects of the integrated schema from the internal (DIAM) aspects. Examples of these internal aspects are given as "network-wide access routes, local database relationships, inter-database relationships, etc. ... (expressed in a form) independent of a specific implementation" [CARD 87, p. 571].

Global query decomposition in UCLA HD-DBMS requires the internal integrated model. Also, portions of the internal integrated model pertaining to specific sites are used in local query processing (which is performed at the query site - the finished, locally processable subqueries are transmitted to the remote sites). In an HDDBMS design in which database integration is defined as a sequence of operators in a formal mapping language, query decomposition can be based on the partitioning of that sequence into subsequences. However, in HD-DBMS a set of formalized operators is not provided for deriving the integrated conceptual (E-R) schema from the conceptual local participant schemata. In [CARD 87] is stated that graphical tools would be used to "paint" E-R schemata and that the

integration process would be semi-automated, but it is unclear as to exactly how the integrated conceptual schema would be derived (note that in [CARD 87], a working HD-DBMS prototype is reported as not yet built). It is also unclear how the integrated internal schema would be specified. Also, conflict resolution is not addressed in [CARD 80] or [CARD 87].

A predicative global query language called ER DML is proposed for the HD-DBMS, and algorithms to translate from SQL, DL/1 and CODASYL DML into ER DML are described as under development in [CARD 87] (a detailed example of SQL to ER DML translation is given). It is pointed out in [CARD 87] that proposed "E-R algebras" are not sufficiently developed for incorporation into the HD-DBMS and that such languages are necessarily more difficult to define than relational algebras since they would have to handle two distinct data model entities as compared to the one entity of the relational model. Besides constraining the choice of query language, the lack of an "E-R algebra" means that database integration cannot be specified algebraically.

Unlike implemented systems, the HD-DBMS is intended to support global updates as well as global queries, and perhaps this is one reason for the relatively complex global data model arrangement. However, it is unclear exactly how global updates would be facilitated by this arrangement.

More 'semantic' models than the basic relational,

functional, and entity-relationship models are sometimes advocated as better media for database integration. In most cases, extended versions of the E-R model are advocated [BATI 84], [BATI 86], [NAVA 84]. These typically include constructs for specifying generalization and/or subset hierarchies and constraints on the connectivity (or cardinality) of relationships. Generally, these models have been used as the basis of "schema integration methodologies" rather than as global data models in specific HDDBMS designs. The methodologies are meant to have applications in centralized database design (where the designer starts with desirable external user views and integrates them to arrive at a single conceptual view) as well as in database integration. They are informal methodologies, explained with graphical manipulations of the extended E-R model diagrams rather than through the application of distinct, formalized operations. Also, very little attention is paid to conflict resolution.

These extended E-R models and the accompanying methodologies provide a high-level way of designing database integration alternatives. Mapping languages, as used in current HDDBMS prototypes, provide a low-level way of implementing the integration mappings. Realistic database integration tasks (as opposed to 'toy' examples) involving large databases clearly cannot be managed entirely in terms of low-level mapping operations, but the high-level methodologies do not provide a precise basis for automated

query processing. Therefore, the two approaches should be regarded as complementary. The high-level methodologies might become the basis for an interactive, semi-automated tool for generating low-level mapping operations. Also, specifications in a more semantic data model might also provide information needed for global updates, but further research is required to ascertain this.

2.6 PROTOTYPE DESIGN APPROACH

A goal of this thesis was to produce a working HDDBMS prototype in which global queries involving at least two local host databases, managed by different DBMSs, would be handled. Thus the design and implementation had to consider practical ways of offering basic functionality rather than focus in great depth on a single difficult issue, such as global updates or concurrency control.

The design has two basic components: a global query and mapping language, and a query processing algorithm. A relational global data model is used, and the global query and mapping language is based on the relational algebra with extensions for database conflict resolution. Although this is not a unique approach, our language is considerably simpler than others encountered in the literature and was designed to be easily processed. Nevertheless, it accommodates all the database conflicts discussed in Section 2.2, as shown in Chapter 4.

The query processing algorithm acts entirely as a front-end to local DBMSs, each of which is expected to be able to support any global query and mapping language operation. It is designed for an environment where sites are joined by high speed local links and where no database statistics (e.g. relation size profiles) are maintained by local DBMSs.

The current design does not support global updates and so does not consider global concurrency and recovery control.

In basic concepts, our design borrows from a number of existing HDDBMS prototypes. However, in the process of developing the design, we have addressed many specific issues which are not discussed in the literature. Also, we have consolidated database conflict types and conflict resolution techniques dealt with separately in other work. The design provides a basic framework for future extensions discussed in Chapter 7.

CHAPTER 3 A GLOBAL QUERY AND MAPPING LANGUAGE

This Chapter discusses the design of an extended relational algebraic language which may be used as a global query and mapping language in an HDDBMS with a relational global model. In particular, the operations supported by this language (referred to hereon as the GQML),the relationship of GQML limitations to HDDBMS implementation, the use of the GQML in database integration, and its accommodation of the schema architecture described in Section 2.1 will be discussed. The formal syntax for the language is given in the Appendix. Specific examples of the application of the GQML to database integration will be given in Chapter 4.

A subset of this language has been implemented as part of the HDDBMS prototype described in Chapter 6.

3.1 OPERATORS

The GQML has the following operators (keywords in bold type):

1. **u** : union
2. **int** : intersection
3. **dif** : difference
4. **div** : division
5. **lim** : limit
6. **lnj** : limited natural join
7. **ren** : rename attributes
8. **onj** : outer natural join
9. **alt** : alteration
10. **grp** : grouping
11. **trc** : transpose row to column
12. **tcr** : transpose column to row

The first six operators (u to lnj) provide relational completeness in the sense of [CODD 72]. They are basic and do not provide capabilities beyond those provided by a typical relational query language. Further capabilities are required for the purposes of database integration, in other words to provide an adequate 'mapping' capability. The ren (attribute renaming) onj (outer natural join), alt(eration), grp (grouping), trc (transpose row to column) and tcr (transpose column to row) operators are provided for this reason.

3.1.1 Basic Relational Operators

The u(nion), int(ersection), dif(ference), and div(ision) operators have the same meaning as the basic relational algebra operators of the same name described in [DATE 86]. However, the notion of union-compatibility which is adopted in this design differs slightly from the definition given in [DATE 86], which says that two relations are union-compatible if there is a one to one correspondence of attributes between the relations, and corresponding attributes are defined on the same domain. In this definition, the corresponding attributes do not have to have the same name. However, in the GQML design, the attributes are also required to have the same name for the relations to be considered union-compatible and, therefore, to be legal operands of the same u, int, or dif operation. This matches the definition of union-compatibility used in the ASTRID

relational algebra language [GRAY 84] and makes it unnecessary to implement an arbitrary rule (which users would have to remember) for assigning names to attributes of the result of a **u**, **int**, or **dif** operation. This also eliminates any ambiguity with regards to determining the correspondence of operand attributes. The GQML can be used to rename attributes (this is described later) and this capability overcomes limitations which would be imposed by the more restrictive definition of union-compatibility.

In the description of the divide operation in [DATE 86] the attributes of the divisor relation need only be defined on the same domains as the corresponding attributes of the dividend relation; matching names are not required. However, the **div** operation requires a match on attribute names as well as domains. This requirement, together with the restriction that no two attributes in a single GQML operand can have the same name, simplifies implementation considerably since it eliminates any ambiguity in determining the correspondence among operand attributes.

Many references, including [ULLM 82], [PIRO 82], and [CODD 72], state that result of a division operation involving an empty dividend relation is itself an empty relation. However, no reference was found which proposes one 'correct' way of handling the case of an non-empty dividend relation divided by an empty divisor relation. In [PIRO 82] two "sensible" alternatives are proposed: (1) that the

result be the projection of the dividend relation on the non-divisor attributes, and (2) that the divide operation be defined only when the divisor is non-empty. The first alternative is assumed in the GQML design since the second alternative would allow the occurrence of errors depending only upon the contents of the local host databases. Such occurrences would be practically unpredictable and unnecessarily disruptive. Also, the semantics of the first alternative are reasonable. For example, it is not necessarily wrong if the query "list customers who have a tab at every branch in the north end of the city" produces a list of all the customers if there are no stores in the north end of the city (this query might be implemented by the division of a projection of a 'TAB' relation by a projection of a 'BRANCHES' relation restricted to branches with a "NORTH" Location - the GQML syntax will be given later). It would be desirable for the user to also know that there are no banks in the north end. However, the GQML as currently designed would not automatically provide this information.

The **u**, **int**, **dif**, and **div** have a simple syntax - the operator keyword followed by the two operands. For example, the union of relations 'REL1' and 'REL2' is indicated by

u REL1, REL2;

Further examples will be given later.

The order of operands is significant in the **dif** and **div** operations. In the **dif** operation the second operand is subtracted from the first. In the **div** operation, the first operation is divided by the second.

The lim(it) operation can be used to specify a selection or a projection or both. In the case of both, the operation behaves as though the projection is made on the result of the selection. For example, consider the following relation:

TAB (CustName, Branch, Balance)

A list of the names of customers with a balance greater than 10000 cents would be obtained with the following:

```
lim TAB where Balance > 10000 attrs CustName;
```

The same selection or projection can be specified separately as follows:

```
lim TAB where Balance > 10000;   (selection)
lim TAB attrs CustName;          (projection)
```

A GQML operation can either be a relation name or another operation. Operations may be nested to any depth and unnecessary spaces, tabs, and newline characters are ignored. The following example of a nested query corresponds

to the division example discussed earlier:

```
div
  lim TAB attrs CustName, Branch;,
  lim BRANCHES where Location = "NORTH" attrs Branch;
  ;
```

Assume that the relation 'BRANCHES' has attributes {Branch, Location} where the attribute 'Branch' in relation 'TAB' is defined on the same domain.

The **lnj** (limit on natural join) operator is used to specify a natural join and, if desired, the equivalent of a **lim** operation on the result, all in the same operation. For the natural join, the operation defines a correspondence between attributes having the same name and defined on the same domain in the separate operands. Tuples having the same values on the corresponding attributes are joined. If there are no corresponding attributes, then the operation behaves as a cartesian product. Duplicate attributes are automatically eliminated from the result, as happens with the natural join operation defined in [DATE 86] and many other sources. A natural join is not allowed between relations containing attributes with the same name but defined on different domains. This restriction pre-empts the occurrence of a result relation with duplicate attribute names. If we want to join relations on attributes with different names or join relations containing similarly named attributes defined on different domains, we can first use the renaming capability of the GQML which will be described later. This is similar to the approach used in the ASTRID

language [GRAY 84].

A natural join between the 'TAB' and 'BRANCHES' relations described earlier would be specified as follows:

```
lnj TAB, BRANCHES;
```

The join attribute would be 'Branch'. The same join followed by a selection on 'Balance' and a projection on the 'CustName' and 'Location' attributes could be specified in one operation as follows:

```
lnj TAB, BRANCHES where Balance > 10000
  attrs CustName, Location;
```

A theta join operation [DATE 86] may be emulated by specifying a **lnj** operation on operands that have no common attributes, and including a **where** clause. For example, consider the following two relations:

```
BOAT (BoatNm, Wt)
TRUCK (TruckNm, Weight)
```

A (TruckNm, BoatNm) relation where for each tuple the corresponding truck weighs more than the corresponding boat can be obtained with the following operation:

```
lnj TRUCK, BOAT where Weight > Wt
  attrs TruckNm, BoatNm;
```

3.1.2 Rename Attributes Operator

The ren operator is used to rename one or more attributes of a relation. Renaming does not change the domain of an attribute or the values associated with it. As an example, consider the relations with schemes Empl(ID, Name, DeptID) and Emp2(EmpNo, Name, DeptNo).

Suppose that there is a requirement to take the union of the two relations and that attributes 'ID' and 'EmpNo' have the same domain, as do attributes 'DeptID' and 'DeptNo'. The corresponding attributes in one or both relations would have to be renamed to the same name before the union. One way of specifying the renaming and the union would be as follows:

```
u
   ren Empl ID to EmpNo, DeptID to DeptNo;,
   Emp2;
```

The result relation would have the same schema as Emp2.

3.1.3 Outer Natural Join Operator

The **onj** operator is basically similar to the outer natural join as defined in [CODD 79]. Informally, the result of an outer natural join of two relations is the union of the natural join of the relations with the set of tuples from both relations that do not participate in the natural join (call these the 'non-join tuples'). Non-join tuples which are not union compatible with the join result are made so by the addition of attributes having null values. These null values would signify 'value at present unknown' if the open world assumption (OWA) is chosen for the database, or 'property inapplicable' if the closed world assumption (CWA) is chosen. The CWA states that the database contains data representing all true facts [GRAY 84], while the OWA allows facts which are not in the database to be true.

In the GQML the **onj** operator differs from the outer

natural join as described above in two ways. First, the values used to 'pad' tuples in order to make them union compatible with the join result are not necessarily 'null', or distinct from 'real world' values in a database. This is so because an onj operation may actually be performed by a local DBMS, as might be the case for any GQML operation in a given HDDBMS implementation, and many DBMSs do not support null values.

The second difference is that the onj operation allows the specification of an additional 'origin' attribute in the result relation. For non-join tuples, this attribute is automatically evaluated to contain the name of the operand relation from which each tuple was taken. For the other tuples the single origin attribute gets a null value or a default value different from either relation name. The user specifies the name of the origin attribute and the system gives it a default domain related to the system's restriction on relation name length.

The outer natural join is an important tool in database integration when the local host databases portray overlapping populations of objects (see Section 4.1). However, in database integration it is often necessary to determine the origin of tuples in an outer join result. In the PRECI Algebraic Language (PAL) used in the PRECI* HDDBMS implementation [DEEN 87] such tuples are identified as originating from a particular operand by the presence of a

null value in an attribute that is not in the schema of that operand. In a HDDBMS where default values may occur instead of null values, the only reliable way of determining the origin of tuples in an **onj** operation result is to examine the value of the origin attribute.

For an example of the application of **onj**, consider the following relations where attribute 'ID' is defined on the same domain in both:

R1 (A	ID)
a1	i3
a2	i5
a3	i2
a4	i6

R2 (B	ID)
b1	i1
b2	i2
b3	i3
b4	i4

The outer natural join of 'R1' and 'R2' with origin attribute 'SOURCE' (indicated as such with the 'ori' keyword) would be specified as follows:

onj R1, R2 ori SOURCE;

The result relation would have the following extension (* represents the default value):

(SOURCE	A	B	ID)
*	a1	b3	i3
*	a3	b2	i2
R1	a2	*	i5
R1	a4	*	i6
R2	*	b1	i1
R3	*	b4	i4

4.1.4. Alteration Operator

The **alt**(eration) operator is a synthesis of the EXT(end) and REP(lace) operators of PAL [DEEN 87]. This operator is used to **drop** attributes from the operand

relation and **add** new attributes to it. The **drop** clause functions as a complementary projection operation; the listed attributes are removed rather than retained. The **add** clause allows the name and domain of one or more new attributes to be specified, and allows the user to specify the value of each new attribute in each tuple either as a constant or as a function of the data in the operand.

For an example of the application of the **alt** operator, suppose that it is required to generate a new relation from the 'TAB' relation described earlier where the 'Balance' attribute is replaced by a "balance in U.S. funds" ('USBal') attribute and a "balance in Canadian funds" ('CanBal') attribute. Suppose also that the current 'Balance' attribute is evaluated in Canadian funds and that 1 cent Canadian is worth 0.8 cents American. This new relation can be generated with an **alt** operation as follows:

```
alt TAB
  drop Balance
  add
    USBal numeric 8 = Balance * 0.8,
    CanBal numeric 8 = Balance
;
```

In this example '**numeric** 8' defines the domain of the new attributes (domains, expressions and predicates will be discussed in more detail in Section 3.2). Note that if the domain of 'Balance' was also '**numeric** 8', then 'CanBal' would essentially be a renaming of the 'Balance' attribute.

A function for evaluating a new attribute may be based on a number of alternatives, depending on the values of one or

more attributes in the operand relation. As an example of this, suppose that it is required to generate a new relation from the 'BRANCHES' relation where the value for the 'Location' attribute is changed to "NORTHEND" or "SOUTHEND" depending on whether the value is "NORTH" or "SOUTH", respectively, in the 'BRANCHES' relation, otherwise the value is the same. The new relation can be defined with an **alt** operation as follows:

```
alt BRANCHES
  drop Location
  add Location char 15 =
    "NORTHEND" if Location = "NORTH" else
    "SOUTHEND" if Location = "SOUTH" else
    Location
;
```

Note that the 'Location' attribute referred to in the function and in the **drop** clause belongs to the operand relation 'BRANCHES' while the 'Location' attribute referred to in the **add** clause belongs to the result relation.

The user can add a new attribute without specifying an evaluation function. In this case, the attribute would be evaluated to null or a system default value in each tuple, depending on the implementation. If, for a given tuple, none of the conditions in an evaluation function are met, and if no default value is specified (the value of attribute 'Location' in the previous example is a specified default) then the value of the new attribute in that tuple will be set to null or a system default value. For example, the following operation will set the value of the new "Location" attribute to null or a system default if the old "location"

attribute value is neither "NORTHEND" nor "SOUTHEND."

```
alt BRANCHES
  drop Location
  add Location char 15 =
    "NORTHEND" if Location = "NORTH" else
    "SOUTHEND" if Location = "SOUTH"
;
```

3.1.5 Grouping Operator

The **grp** operator is similar to the "group_by" operator of the ASTRID language [GRAY 84] in that it behaves as a projection operator combined with extension. Consider the following relation schema and the **grp** operation on the corresponding relation:

```
EMP ( EMPNO   EMPNAME   DEPTNM   DEPTLOC   SALARY )

grp EMP by DEPTNM, DEPTLOC
        add AVGSAL numeric 9 = agg_avg(SALARY),
            MINSAL numeric 9 = agg_min(SALARY),
            MAXSAL numeric 9 = agg_max(SALARY)
;
```

The result of the operation is a relation with attributes 'DEPTNM', 'DEPTLOC', 'AVGSAL', 'MINSAL', and 'MAXSAL' where the last three attributes hold values for the average, maximum, and minimum salary values in tuples with the same 'DEPTNM' and 'DEPTLOC' values.

The projection attribute list (consisting of, in this case, 'DEPTNM' and 'DEPTLOC') specifies that tuples having the same values in those attributes are to be part of the same 'group'. If no **add** clause is included in the operation, then the result is simply a projection on the listed attributes.

New attributes may be specified as in the **alt** operation and their values specified as 1) constants, 2) user defined or system defined non-aggregate functions of the projection list attribute values, or 3) system defined aggregate functions. In the result there is one tuple for each 'group' as defined by the projection attribute list, and new attributes are evaluated separately for each such tuple. If no projection attribute list is given (i.e. if the **by** clause is omitted) then the result relation has only the new attributes and one tuple.

Aggregate functions consider together all tuples of a group. The aggregate functions included in the GQML depends on the implementation, but a useful minimum collection would probably be **agg_avg**, **agg_min**, **agg_max**, **agg_sum**, **agg_count**, **agg_any**, and **agg_all**. The first four have obvious meanings. The **agg_count** function would not take an argument and would return the number of tuples in the group. The **agg_any** and **agg_all** functions would have a **logical** argument (see Section 3.2) and would return **TRUE** if the argument was true for any or all tuples in the group, respectively, otherwise they would return **FALSE** [GRAY 84].

The use of **grp** with no projection attribute list and new attributes defined with aggregate functions is a way of obtaining summary statistics of a relation as a whole. For example, if in the previous example the **grp** operation had no **by** clause, the result relation would only have one tuple

with attributes 'AVGSAL', 'MINSAL', and 'MAXSAL' holding average, minimum, and maximum salary values for the entire 'EMP' relation.

3.1.6 Transpose Operators

The **trc** (transpose row to column) and **tcr** (transpose column to row) are basically similar to the operators in PAL having the same name. A **trc** operation transforms a relation of the form

R (a1, a2, ..., am, c1, c2, ..., cn)

into a relation of the form

R (a1, a2, ..., am, b, c)

by changing values for 'c1' to 'cn' into values for 'c' and adding attribute 'b' which has integer values to indicate sequencing. Attributes 'c1' to 'cn' must be defined on the same domain, which automatically becomes the domain for 'c'. A **tcr** operation performs the inverse transformation. It is permissible for there to be any number of 'a' attributes, including zero. In the PAL version of these operations, only one 'a' attribute is allowed [DEEN 87].

For an example of the application of the trc and tcr operators, consider the following relations:

SALES1 (EmpID EmpNm P1Sales P2Sales P3Sales)

EmpID	EmpNm	P1Sales	P2Sales	P3Sales
100	"Smith"	53	49	51
200	"Jones"	60	52	55

SALES2 (EmpID EmpNm Period Sales)

EmpID	EmpNm	Period	Sales
100	"Smith"	1	53
100	"Smith"	2	49
100	"Smith"	3	51
200	"Jones"	1	60
200	"Jones"	2	52
200	"Jones"	3	55

The 'SALES2' relation can be generated from the 'SALES1 relation using

```
trc SALES1 row P1Sales, P2Sales, P3Sales
           to_col Sales
           seq Period;
```

The 'SALES1' relation can be generated from the 'SALES2' relation using

```
tcr SALES2 col Sales
           to_row P1Sales, P2Sales, P3Sales
           seq Period;
```

In the trc operation, the order in which attributes are listed in the row clause is important because this determines the result relation correspondence of values for the attributes specified in the to_col and seq clauses. Similarly, the order in which attributes are listed in the to_row clause of the tcr operation is associated with the operand relation values of the attributes specified in the seq clause.

In the trc operation the attribute specified in the seq clause is given a domain with a numeric type by the system

and is evaluated to successive integer values, starting with 1, in the result relation. Similarly, the **tcr** operation requires that the attribute specified in the **seq** clause has a numeric type domain in the operand relation. In order for tcr to produce a valid result, the values of this attribute must be restricted to range from 1 to the number of attributes specified in the **to_row** clause. Additional requirements for valid results are that the attributes in a trc operand which are not listed in the **row** clause (i.e. the 'a' attributes) must constitute a key of the relation (if there are no 'a' attributes then there must only be one tuple in the operand), and that the sequencing attribute in a tcr operand must be a prime attribute.

In the operand of a **tcr** operation we might normally expect each set of values for the 'a' attributes to be associated with each permissible value of the sequencing ('b') attribute. For example, in 'SALES2' 'a' attribute values <100, "Smith"> are associated with all permissible 'Period' values. Suppose that this is not the case and that one or more combinations of values for the 'a' attributes which occur at least once in the operand relation are not associated with each permissible value of the sequencing attribute. For example, consider a tcr operand equal to 'SALES2' with the first tuple missing, so that 'a' attribute values <100, "Smith"> are not associated with the permissible sequencing attribute value of 1 ({'EMPID', 'EMPNUM'} are the 'a' attributes and 'Period' is the

sequencing attribute). In the result relation, the tuple in which such a combination of 'a' attribute values appears will have a null or default value (depending on the implementation) for the row (or 'c') attributes corresponding to the sequencing attribute values that do not occur in the same operand tuple as those 'a' attribute values. So, the earlier **tcr** operation on our modified version of the 'SALES2' relation will produce a result similar to 'SALES1' except that the 'P1SALES' attribute will have a null or default value in the result relation tuple in which values <100, "Smith"> occur for attributes <'EMPID', 'EMPNM'>.

If a potential **tcr** operand has a sequencing attribute that is not defined on a numeric domain or does not have the required integer values (e.g. the attribute has date values), then it would be necessary to perform an **alt** operation on this relation to change to sequencing attribute before the **tcr** operation. Likewise, the sequencing attribute in a **trc** operation result relation could be replaced by a new, more appropriate sequencing attribute with an **alt** operation.

3.2 DOMAINS, EXPRESSIONS AND PREDICATES

Each attribute in a global relation must be defined on a domain, which is an identifiable set of data values. The concept of domain used here is similar to the concept of

data type in programming languages, but the word 'domain' will be used for consistency with relational data model terminology. For the purposes of discussion we will consider three basic domain types: character strings, numbers, and a domain containing 'true' or 'false'. Factors limiting the domains which can be included in the GQML will be discussed in Section 3.3. A character string domain is indicated with the **char** keyword followed by a length integer of 1 or greater. A number domain is indicated with the **numeric** keyword followed by a length integer of value 1 or greater and, optionally, a scale integer separated from the length integer by a period, as in '**numeric** 10.2'. To specify an integer number domain accommodating n digits and a sign the **numeric** keyword would be used with a length value of n, as in '**numeric** 10'. To specify a 'real' domain accommodating n digits of which m are fractional, and a sign, the **numeric** keyword would be used with a length value of n+1 (the extra place is for the decimal point) and scale value of m. A true/false domain is indicated with the **logical** keyword.

Expressions and predicates are required in **add** clauses of **alt** and **grp** operations, and predicates are required in **where** clauses of **lim** and **lnj** operations. In the context of the arithmetic and logical operations required to build expressions and predicates, atomic operands are either literals or attribute values (represented by attribute names). The common arithmetic operations (addition, subtraction, multiplication, division, exponentiation) may

be performed with **numeric** operands. Operands of **numeric** domains may be compared with one another using >, <, >=, =<, =,and <> to yield a truth value, as can operands of **char** domains (in the latter case lexical ordering is the basis of comparison). Operands of the **logical** domain, including predicates, may be compared with each other using '='. The logical operators **and, or** and **not** may be used to construct predicates. The GQML provides **FALSE** and **TRUE** logical constants.

The definition of attributes on domains prevents meaningless comparisons; for example the comparison of a **numeric** value with a **char** value. It also provides information required to build data structures for storing intermediate and result relations when evaluating a GQML query.

3.3 GQML LIMITATIONS RELATED TO HDDBMS IMPLEMENTATION

The implementation of a GQML for a HDDBMS differs from the implementation of a query language for a centralized DBMS. In the centralized case, the features of the query language are matched with a single set of data processing capabilities. In the case of a HDDBMS, the pre-existing local DBMSs may have different data processing capabilities and may vary in the kinds of queries that they can accommodate. Furthermore, they may not all support certain GQML operations on relations and data values which are

required for the purposes of database integration. For example, a special-purpose file management system may not support the equivalent of a join between global relations or the DML of a simple local DBMS may not have function for converting a value from a numeric representation to a character string representation. These differences are more important for HDDBMSs designed to delegate greater amounts of data processing to the local DBMSs.

Support for an operation implies support for the domains of the data values involved in the operation. For example, if a selection predicate involves a comparison of an attribute and a literal from a non-atomic 'date' domain then the local DBMS that processes the corresponding lim operation would have to correctly model a 'date' domain and comparisons involving 'date' domains. Simply converting date values to character strings and using lexical ordering as the basis of comparison would not be correct and therefore would not constitute support of the 'date' domain ("05/11/86" precedes "15/11/76" if these values are interpreted as character strings, but the reverse is true if they are interpreted as dates with a day/month/year format). A character string representation of values from a non-character string domain is useful only for the purposes of data storage and display.

In one extreme, the HDDBMS may be designed as a purely 'front-end' system so that all data processing is actually

performed by local DBMSs. If the HDDBMS does not consider differences in local DBMS capabilities in planning a global processing strategy then any local DBMS must be capable of supporting, in some way, any GQML operation. This means that the GQML operations which may be implemented would be limited to those that are supported by all local DBMSs.

In an HDDBMS of the type described above, any data processing capabilities which are not shared by all local DBMSs could still be exploited locally. For example suppose that the same property is modelled with a single 'Date' attribute in database A (managed by DBMS A) and with separate 'Day', 'Month' and 'Year' attributes in database B (managed by DBMS B). A function for creating a single 'Date' value from separate 'Day', 'Month', and 'Year' values, and functions for extracting 'Day', 'Month', and 'Year' values from a single 'Date' value would be a useful part of the GQML. Suppose further that such functions are supported by DBMS A only so that they must be excluded from the GQML. The transformations that such functions perform could simply be neglected, thus compromising the quality of the integrated schema. Alternatively, the 'Date' field decomposition could be performed locally as part of the mapping of the database A schema from the local host version to the translated local host version. Thus, in the translated schema version of database A, the property in question would be represented as separate 'Day', 'Month', and 'Year' attributes and would therefore be union compatible with the same property in the

translated schema version of database B. The disadvantages of this approach as compared to having the date functions in the GQML is that the database integration task is distributed to one more mapping level and the ways of expressing an integrated schema are limited. In the above example, an integrated schema where there is a single 'Date' attribute might be preferable from the database A user's viewpoint. A further level of mapping to a separate site A external schema in which DBMS A capabilities are again applied locally would be required, this time to combine the 'Day', 'Month', and 'Year' attributes of the integrated schema into a single 'Date' attribute.

The problems of a 'front-end' system as described above could be lessened if the HDDBMS was designed to assign an operation only to those local DBMSs capable of supporting it when planning a global processing strategy, thus using the strengths of some local DBMSs to compensate for the weaknesses of others. This would, on average, entail more data movement than in a system with equally capable local DBMSs since data would have to be transferred from a site with an inadequate local DBMS to a site where a more capable DBMS resides. However, the GQML operations which could be implemented would be the union of those supported by all local DBMSs.

At the other extreme of HDDBMS design, there would be a 'super DBMS' at each site that would accommodate all domains

and operations of any local DBMS as well as those required for database integration. These 'super DBMSs' would take over all data processing. At each site there would be local DBMS-specific modules for mapping between local host and translated local host schemata and, if required, for providing a user interface similar to that of the local DBMS. The capability of the 'super DBMS' to subsume any local DBMS would be enhanced if it could be extended with user defined domains and operations, perhaps in the manner described in [LINN 88]. The only difference between this design and that of a conventional closed DDBMS is that the local host schemas have been designed independently and the local host databases may be stored in different data structures.

Designing an HDDBMS with 'super DBMSs' as described above is probably the most conceptually simple approach since local DBMS heterogeneity is made irrelevant as early as possible and the necessary interfaces to local DBMSs would be basic. However, implementing such a system would mean implementing many functions that already exist in the local DBMSs. Even if a library of suitable DBMS software tools is available to facilitate implementation, the size of the HDDBMS would be larger than one implemented as a front-end. This would be of special concern in microcomputer environments with relatively limited primary and secondary memory. The 'super DBMS' could completely replace the original local DBMS, and this would be desirable if the

HDDBMS is expected to manage concurrent global and local transactions. However, it would then be necessary to provide a local interface to the 'super DBMS' that mimics the original local DBMS and accommodates old application programs and users who wish to bypass the HDDBMS.

In cases where no local DBMS can support a required database integration operation, it would be necessary to augment the aggregated processing power of the local DBMSs at least for this operation. A 'middle-of-the-road' approach somewhere between the extremes described above may be sufficient and most practical. For example, a 'super DBMS' may be connected to the system at one or more sites but would be used only when necessary, or when it is more efficient to do so. This is the approach taken in the design of the Multibase system [LAND 82]. However, the added processing power may not have to be as comprehensive as this. For example, a special system may be implemented for decomposing non-atomic attributes into multiple separate attributes (or, conversely synthesizing multiple attributes into a single attribute), for computing mathematical functions such as mean and square root, for converting values from one domain to another (e.g. numeric to character string), and for computing aggregate functions (e.g. the average of an entire column of values). Such a system might run directly on global format data.

3.4 DATABASE INTEGRATION WITH THE GQML

Database integration with the relational model, for the purposes of read-only requests, may be regarded as consisting of two basic steps: (1) resolving schema conflicts among a collection of base relations, if necessary, and (2) merging multiple conflict-free relations into single relations, if necessary, while resolving data conflicts in the process. The use of "if necessary" here refers to the fact that a collection of base relations as is may be a satisfactory database. Structural schema conflict resolution and relation merging may be performed differently at different sites to arrive at different integrated schemas. Examples of these activities, as performed with the GQML, will be presented in Chapter 4.

As implied above, the starting point of database integration at a given site in a HDDBMS using a relational global model is the knowledge that there are base relations accessible from that site whose semantics allow them to be considered as building blocks of a single database. Database integration need not be concerned with how or from where these base relations are obtained. The design of the GQML recognizes this by requiring a simple declaration of each base relation.

A base relation declaration in the GQML specifies a relation schema and is an assertion that in the integrated system the extension of the corresponding relation exists.

For example, the base relation 'TAB' could be declared as follows:

```
base TAB
    key CustName char 15,
    Branch char 10,
    Balance numeric 6;
```

In this example, 'CustName' is identified as a key attribute.

In the HDDBMS prototype implementation, base relation declaration also includes data location information; however, this was done as a implementation shortcut and is not an essential aspect of the GQML. This is discussed further in Chapter 6.

With the GQML, database integration is performed by specifying a mapping from base relations to higher level views or virtual relations which represent the integrated database. This mapping consists of a series of GQML operations. User queries can be posed on these virtual relations with the GQML. Any GQML operation can be used for either view mapping or for queries.

To distinguish between a virtual relation and a query result relation, "==" is used in the former case to associate a relation name with an operation (or a series of nested operations), and ":=" is used in the latter case. In the following example, 'RESULT' is the result of a query on virtual relations 'R1' and 'R2':

```
R1 == lim TAB attrs CustName, Branch;

R2 == lim BRANCHES
        where Location = "NORTH" attrs Branch;

RESULT := div R2, R1;
```

Note that the same query can be expressed as follows:

```
RESULT := div
            lim TAB attrs CustName, Branch;,
            lim BRANCHES where Location = "NORTH"
              attrs Branch;
          ;
```

Any relation name or attribute name used in a GQML operation may be prefixed by a special symbol in order to avoid conflicts with GQML keywords. This is useful since base relation schemata represent pre-existing data and will not have been designed with the GQML syntax in mind. Consider the following operation where an underscore is used as the prefix symbol:

```
alt _lim drop _add add new char 5;
```

The operand is assumed to be a relation named "lim"; the leading underscore would be stripped from the name after it has been recognized. If the name "lim" is used without an underscore, then the interpreter or compiler would attempt to parse a nested **lim**(it) operation and eventually recognize a syntax error. Similarly, the underscore before "add" specifies an attribute named "add" rather than the **add** keyword.

The essential difference between the GQML and other

relational DMLs with a view definition capability, (e.g. some implementations of SQL), is that some GQML operators are included primarily for the purposes of integrating multiple pre-existing databases, namely **onj**, **ren**, **alt**, **trc** and **tcr**. The GQML could be used for conventional single database view definition in a centralized DBMS.

3.5 THE GQML AND THE REFERENCE HDDBMS SCHEMA ARCHITECTURE

The schema levels of the reference HDDBMS schema architecture described in Chapter 2 can be accommodated by the GQML by creating corresponding sets of virtual relation definitions and base relation declarations. Note, however, that the GQML cannot support a distinct external schema level unless the external schema is relational or tabular. If not, a separate relational/external mapping facility would be required. Similarly, a local host schema cannot be described with GQML local base relation declarations unless the local database model is relational or tabular. If not, a separate relational/local model mapping facility would be required. The development of such mapping facilities is beyond the scope of this thesis and is a topic for further work (see Chapter 7).

Figures 3.1 to 3.5 illustrates a simple database integration example, using the GQML, for two hypothetical databases at sites 1 and 2. Figure 3.6 shows the virtual relations belonging to each schema level. Local host and

external schema levels are not shown; however, if we assume that the databases at both sites are relational, then the translated local host schemata (Figure 3.1) match the local host schemata, and the integrated schema (Figure 3.5) can also serve as the external schema.

Both databases represent employees and departments. At site 1 the 'WORKS' relation represents a many-to-many relationship between employees and departments, while at site 2 the relationship is one-to-many and is implemented by including the department key as a foreign key in the employee relation. In the mapping from the translated local host schemata to the local participant schemata (Figure 3.2), attribute name conflicts are resolved and a 'CITY' attribute with a constant value ("MONTREAL" or "TORONTO", depending on the site) is added to each of the two relations representing departments in order to resolve an implied data conflict. The global schema consists of the collection of local participant relation schemata from each site (Figure 3.3). In the mapping from the global schema to the integrated schema (Figure 3.4), an employee relation is defined in which employees who are represented at both sites have a total salary which is the sum of their two individual salaries. Other employee data is taken from site 2 where possible. The relationship conflict is resolved by defining a many-to-many relationship between departments and employees via the 'WORKSIN' relation. Conflict resolution techniques are discussed in more detail in Chapter 4.

For the sake of brevity both sites have the same integrated schema in the example. However, different integrated schemata may have been created by using a different mapping from the global schema at each site.

Site 1:

```
base EMP                        base DEPT
  key ID char 5,                  key DID char 5,
  LNAME char 12,                  NAME char 15,
  FNAME char 10,                  BLDG char 15;
  SAL numeric 6;

base WORKS
  key DID char 5,
  key ID char 5,
  START date;
```

Site 2:

```
base EMPS                       base DEPTS
  key EMPNO char 5,               key DNO char 5,
  DNO char 5,                     DNM char 15,
  SNM char 15,                    LOC char 15;
  CNM char 15,
  BIRTHDATE date,
  SALARY numeric 8;
```

Figure 3.1 translated local host schemata

```
Site 1:

     EMP1 == ren EMP attrs
               ID to EMPID,
               LNAME to R1LSTNM,
               FNAME to R1FSTNM,
               SAL to R1SALARY;

     DEPT1 == alt DEPT
                drop DID, NAME, BLDG
                add
                  key DEPTID char 5 = DID,
                  DEPTNM char 15 = NAME,
                  STREET char 15 = BLDG,
                  CITY char 15 = "MONTREAL";

     WORKS1 == ren WORKS attrs
                 ID to EMPID,
                 DID to DEPTID;

Site 2:

     EMP2 == ren EMPS attrs
               EMPNO to EMPID,
               SNM to R2LSTNM,
               LNM to R2FSTNM,
               SALARY to R2SALARY,
               DNO to DEPTID;

     DEPT2 == alt DEPTS
                drop DNO, DNM, LOC,
                add
                  key DEPTID char 5 = DNO,
                  DEPTNM char 15 = DNM,
                  STREET char 15 = LOC,
                  CITY char 15 = "TORONTO";
```

Figure 3.2 mapping from translated local host schemata to local participant schemata

(Site 1 local participant schema)

```
base EMP1
  key EMPID char 5,
  R1LSTNM char 15,
  R1FSTNM char 15,
  R1SALARY numeric 8;

base DEPT1
  key DEPTID char 5,
  DEPTNM char 15,
  STREET char 15,
  CITY char 15;

base WORKS1
  key EMPID char 5,
  key DEPTID char 5,
  START date;
```

(Site 2 local participant schema)

```
base EMP2
  key EMPID char 4,
  DEPTID char 5,
  R2LSTNM char 15,
  R2FSTNM char 15,
  R2SALARY numeric 8,
  DEPTID char 5,
  BIRTHDATE date;

base DEPT2
  key DEPTID char 5,
  DEPTNM char 15,
  STREET char 15,
  CITY char 15;
```

Figure 3.3 global schema

```
TEMP == alt EMP2 drop DEPTID;

EMPLOYEE == alt
                onj EMP1, TEMP1 ori ORIG ;
              drop R1FSTNM, R2FSTNM, R1LSTNM,
                   R2LSTNM, R1SALARY, R2SALARY,
                   ORIG
              add
                FIRSTNAME char 15 =
                   R2FSTNM if ORIG = "TEMP1" else
                   R1FSTNM,
                LASTNAME char 15 =
                   R2LSTNM if ORIG = "TEMP1" else
                   R1LSTNM,
                TOTSALARY numeric 10 =
                   R1SALARY if ORIG = "EMP1" else
                   R2SALARY if ORIG= "TEMP1" else
                   R1SALARY + R2SALARY;

WORKSIN == u
             lim EMP2 attrs DEPTID,EMPID;,
             WORKS1
          ;

DEPTMNT == u DEPT1, DEPT2;
```

Figure 3.4 mapping from global schema to integrated schema (used at both sites)

```
base EMPLOYEE
  key EMPID char 5,
  FIRSTNAME char 15,
  LASTNAME char 15,
  TOTSALARY numeric 10,
  BIRTHDATE date;

base DEPTMNT
  key DEPTID char 15
  DEPTNM char 15,
  STREET char 15
  CITY char 15;

base WORKSIN
  key EMPID char 5,
  key DEPTID char 5,
  START date;
```

Figure 3.5 integrated schema

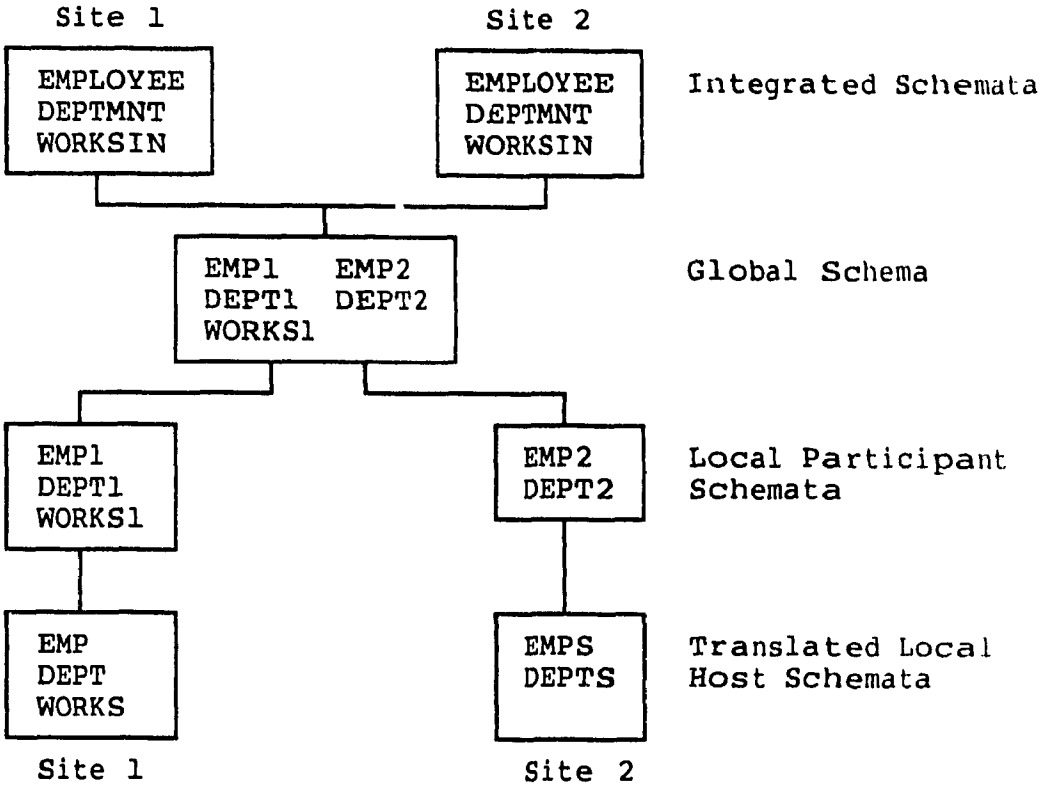


Figure 3.6 virtual relations and schema levels

Note that at a given site, each base relation declaration corresponds to a virtual relation definition unless that base relation in the translated local host schema (in which case it corresponds to stored data at that site), or in the global schema and not in the local participant schema (in which case it corresponds to remote data). Let us refer to a base relation without a corresponding virtual relation definition at a given site as a 'terminal relation' with respect to that site.

Base relation declarations for non-terminal relations are superfluous since the relation scheme of a non-terminal

relation can be derived from the corresponding mapping to terminal relations. In Figures 3.1, 3.3, and 3.5 a set of base relation declarations is given for each schema level simply to make the schema levels plain, rather than to illustrate how a user would actually input mapping specifications. An implementation for mapping specification is described in Section 6.2.3.

CHAPTER 4 CONFLICT RESOLUTION AND RELATION MERGING TECHNIQUES

In this Chapter basic GQML relation merging and conflict resolution possibilities are explained and illustrated with examples. The techniques discussed here require considerable user interaction, just as conventional database design does. The development of database integration design aids is a topic for further work (see Chapter 7).

Relation merging is addressed first in order to provide a context for the discussion of conflict resolution.

4.1 RELATION MERGING

In this Section, the conflict-free relations will be referred to as base relations for convenience. As an alternative to discriminating between 'entity' relations and 'relationship' relations, the word 'object' will be used to refer to anything that is represented by a relation.

In the simplest case, two base relations represent the same class of objects, and are disjoint. An integrated schema relation can be created simply through the union of the base relations (necessary joins between global relations would have already been performed to resolve aggregation and data clustering conflicts). The effect of relation merging in this case is not to create a new object class, but to

consolidate the distributed population of the same object class, as is typically done in closed DDBMSs [CERI 84].

In many cases, two base relations representing the same class of objects may not be disjoint and consequently there is a possibility of data conflict among intersecting tuples. In these cases, it is necessary to identify the intersecting tuples in order to resolve the data conflict. The onj operation provides a way of doing this in the process of merging the base relations (this is why data conflict resolution is paired with relation merging in this discussion). Consider the following base relations:

EMP1 (	EMPID	SAL1)
	100	53
	200	46
	300	50
	500	42

EMP2 (	EMPID	SAL2)
	100	53
	200	50
	400	49
	600	47

Suppose that it is known that the same EMPID value in the two relations means that the corresponding tuples actually refer to the same employee and that the SAL1 and SAL2 attributes, which represent employee salary, should have identical values for those tuples. As is readily seen, the SAL1 and SAL2 attributes do not always have the same value for tuples with the same EMPID value, so there is a data conflict.

The operation 'onj EMP1, EMP2 ori ORIG;' would produce the following result:

ORIG	EMPID	SAL1	SAL2
*	100	53	53
*	200	46	50
EMP1	300	50	*
EMP2	400	*	49
EMP1	500	42	*
EMP2	600	47	*

In this result, the data conflicts are confined to the tuples in which the value of the ORIG attribute is neither 'EMP1' nor 'EMP2'. Call these the 'join tuples'. Note that if both EMP1 and EMP2 used 'SALARY' as an attribute name instead of 'SAL1' and 'SAL2' respectively, it would have been necessary to rename the 'SALARY' attribute in the two relations to different names before the onj operation, thereby deliberately introducing a naming conflict (synonym).

One way of resolving the data conflict would be to use the value from the more credible source, if it can be determined. Suppose that the 'EMP1' relation is deemed to be the more credible source, the following series of operations would produce the required result from 'EMP1' and 'EMP2' assuming that a useable 'SALARY' domain is 'numeric 8':

```
alt
    onj EMP1, EMP2 ori ORIG;
  drop ORIG, SAL1, SAL2
  add SALARY numeric 8 =
    SAL2 if ORIG = EMP2 else SAL1
;
```

the result relation would be

(EMPID	SALARY)
100	53
200	46
300	50
400	49
500	42
600	47

Note that the use of "SAL1" and "SAL2" synonyms is not visible in the integrated schema.

The same approach can be applied in a more elaborate manner. For example, the average of the conflicting values may be used for the join tuples (this would be more appropriate when dealing with measurement error in a attribute such as 'Distance'), or the maximum or minimum value may be used. These solutions would be more conveniently specified using built-in functions such as 'avg', 'min', and 'max' if these are included in the GQML implementation.

An example of another approach would be to use the value from 'EMP1' where possible for a tuple with an 'EMPID' value within a certain range. However, it becomes hard to imagine the rationale for such a computation.

In many cases it may be impossible to determine the most credible source with satisfactory confidence, devise a satisfactory function of the conflicting values, or base a decision on non-conflicting attribute values. In such cases, the best approach may simply be to keep the conflicting value attributes ('SAL1' and 'SAL2' in the above example)

and let the user make the decision as to how to use the data.

In [DAYA 84] it is observed that recorded facts which are expected to be the same in different databases may be different because one database is simply more up-to-date than the other. For example, the salary of the same employee recorded in two databases may vary because the employee has received a raise and only one database has been updated accordingly. This is classified as a data conflict in [DAYA 84], but it is more correctly classified as an implied data conflict because both salary values are correct with respect to the moment in time that each database is supposed to represent.

The associated time data may be implied by the operational context of each database, but is not actually recorded (of course, time data could be recorded, but we assume the presently more common situation where they are not, and where old data is simply discarded). However, with respect to resolution, this type of discrepancy is most practically treated as a data conflict where the value for the integrated schema relation would simply be extracted from the most up-to-date database in the same way that another value may be extracted from the most credible (with respect to error) database. At different times either of the two databases may be most up-to-date (just as reliability may vary). In such cases other rules would have to be

followed such as picking the largest value (probably reasonable in the salary case), or both values would have to be used.

In the cases described so far, mergeable base relations represented the same object classes. It is also possible for base relations to represent subclasses of a common generic object class. As with the previous cases, multiple base relations may represent either disjoint sets of objects or intersecting sets. For example, separate base relations representing managers, secretaries, and technicians, all of whom are employees, would probably be disjoint while those representing employees who are students and employees who are politicians are easily imagined to be intersecting [TEOR 86]. However, unlike the previous cases, the separate base relations may have different attributes without implying unresolved missing data conflicts since they represent different object classes each of which is justified in having its own properties in addition to the common properties. For example, "typing speed" might be an applicable secretary property, but is unlikely to be a manager property.

The subclass base relations could simply be used in the integrated schema without merging, or it may be desirable to create a relation representing the generic class ('Employee' in the above example). Only the properties common to all subclasses are applicable to the generic class. Therefore,

for disjoint subclasses a generic class relation may be created by taking the union of the subclass base relations after they have been projected on the common attributes. For non-disjoint subclasses, the possibility of data conflicts on common attribute values exists, and a similar approach to that described earlier for merging non-disjoint relations representing the same object class is required to merge the subclass base relations, projected on common attributes, into a generic class relation.

4.2 SCHEMA CONFLICT RESOLUTION

The GQML can be used to resolve the structural schema conflicts discussed in Chapter 3. The discussion of conflict resolution techniques will be organized by categories of transforms (sequences of one or more GQML operations) rather than by conflict type since the same type of transform may be used to resolve different conflicts. The transform categories are as follows:

1) attribute add/drop/modify
2) fragmentation redefinition
3) transposition
4) tuple grouping
5) true/false

4.2.1 Attribute Add/Drop/Modify

This type of transform involves any combination of

adding, dropping, and modifying the attributes of a single relation without changes to other relations. Also, the relation before and after the transformation has the same number of tuples, unless attribute dropping results in duplicate tuples (which are eliminated). Attribute add/drop/modify transformations may be used to resolve name (field and data), field, value scale, implied data, derived data, and missing data conflicts.

4.2.1.1 Name Conflict Resolution

A name conflict between two attributes (the relational version of a field-name conflict) can be directly resolved with the **ren** operator. For a homonym-type conflict, one or both attributes are renamed so that both have the same name, for a synonym-type conflict one or both attributes are renamed so that the two have different names. The **ren** operator is basically a convenience since attribute renaming may also be performed with the **alt** operator by replacing one attribute by another having the same domain but a different name and giving the added attribute the values of the dropped attribute.

To resolve a data-name conflict between the values of two attributes with a character string domain, one or both of the attributes would be replaced by an attribute having the same name and domain but evaluated so that the misnamed data items are replaced by those with correct names. The **alt** operator is capable of renaming attributes and data

simultaneously.

4.2.1.2 Field Conflict Resolution

To resolve a field-type (or domain) conflict, the **alt** operator would be used to replace an attribute with another attribute having the same name but a different domain, and then to assign the values of the old attribute to the new. For example, suppose that two relations 'EMP1' and 'EMP2' representing employees have a 'SALARY' attribute, but that in 'EMP1' this attribute has a '**numeric** 8' domain (an 8 digit integer), while in EMP2' it has a '**numeric** 10' domain (a 10 digit integer). This difference in domain alone prevents the two relations from being union compatible. This conflict could be resolved by changing the '**numeric** 8' domain to '**numeric** 10' as follows:

alt EMP1 **drop** SALARY **add** SALARY **numeric** 10 = SALARY;

In many cases data-type conversion would be necessary, for example in order to assign values from a numeric domain to a character string domain. This may be implemented by making type conversion functions available to the user, or with a more sophisticated implementation it may be done automatically. A value from one domain may not be representable in another domain in which case a null, default or special error value would be assigned to the new attribute.

In order to resolve single-versus-multiple field conflicts where a single field in one database holds the encoded data of several fields in another database, the GQML would have to include special functions of data values. For example, there might be a '**make_date**()' function that would convert three numbers into a date domain value or a '**get_year**()' function that would extract a number representing the year from a date value.

An **alt** operation would be useful in cases where a single attribute of a relation has alternate meanings which are represented by multiple attributes of another relation (the definition of an attribute with alternate meanings may be a questionable practice in relational database design, but nevertheless it may occur). Consider the following relations:

EMP1 (EMPID	JOBTYPE	COMMIS	OVERTM)
100	SALES	52	*
200	OFFICE	*	15
300	SALES	*	20
400	OFFICE	60	*

EMP2 (EMPID	JOBTYPE	COMMOVER)
100	SALES	52
200	OFFICE	15
300	SALES	20
400	OFFICE	60

Both relations contain the same information - the commission or overtime earned by four employees. It is assumed that if an employee has an 'OFFICE' type job then she cannot earn a commission, and that if she has a 'SALES' type job then she cannot earn any overtime. The only difference between the two relations is that 'EMP2' uses one attribute, 'COMMOVER'

to represent the same data as 'COMMIS' (commissions) and 'OVERTM' in 'EMP1'.

A relation similar to 'EMP2' can be made from 'EMP1' using the following operation:

```
alt EMP1
  drop COMMIS, OVERTM
  add COMMOVER numeric 3 =
    COMMIS if JOBTYPE = "SALES" else
    OVERTM if JOBTYPE = "OFFICE"
;
```

A relation similar to 'EMP1' can be made from 'EMP2' using the following operation:

```
alt EMP2
  drop COMMOVER
  add
    COMMIS =
      COMMOVER numeric 3 if JOBTYPE = "SALES",
    OVERTM =
      COMMOVER numeric 3 if JOBTYPE = "OFFICE"
;
```

Note that in these examples, attribute "JOBTYPE" is required to determine whether commission or overtime is applicable to a particular employee. A similar case can be imagined where the same information is encoded in another attribute. For example, odd employee identifier values may be used for sales personnel while even values may be used for office personnel. Special functions would have to be present in the GQML implementation to directly interpret such attribute values. Alternatively, auxiliary data may be added to the system to map the values to their encoded meanings. Suppose that in the above example the 'EMPx' relations have no 'JOBTYPE' attribute and that instead job-

type is encoded in the employee identifier. An auxiliary relation with attributes {JOBTYPE, EMPID} containing tuples that match each employee identifier with the corresponding job-type could be joined with the 'EMPx' relations to provide the necessary information for the **alt** operations.

The following relations provide an example where a single attribute in one relation ('USER' in 'CARS2') encompasses the domains of multiple attributes in another relation ('DEPTNM' and 'EMPNO' in 'CARS1'). Assume that 'USER' and 'DEPTNM' are **char** 8 and 'EMPNO' is **char** 3. The 'USER' attribute encompasses identifiers for department or individual employee users, while the 'DEPTNM' and 'EMPNO' attributes are limited to department identifiers and employee identifiers respectively (these restrictions might be enforced by the host DBMS).

CARS1 (CARNUM, USERTYPE, DEPTNM, EMPNO)

CARNUM	USERTYPE	DEPTNM	EMPNO
100	DEPT	BLDGINSP	*
200	EMP	*	412
300	DEPT	ASSMNT	*
400	EMP	*	320
500	DEPT	SERVICES	*
600	EMP	*	215

CARS2 (CARNUM, USER)

CARNUM	USER
100	BLDGINSP
200	412
300	ASSMNT
400	320
500	SERVICES
600	215

A straightforward **alt** operation can create a relation similar to 'CARS2' from 'CARS1':

```
alt CARS1 drop USERTYPE, DEPTNM, EMPNO
          add USER char 8 =
            EMPNO if USERTYPE = "EMP" else DEPTNM;
```

The reverse transformation (from 'CARS2' to 'CARS1' is not as easy since there has to be some way of determining whether the value of the 'USER' attribute for a particular tuple represents a department identifier or an employee identifier. The solution would be simple if the GQML implementation had functions for directly testing a value for domain membership (e.g. '**is_a_dept_id**()'). However, a solution may be possible even without such functions. Suppose that relations 'EMPS (USER)' and 'DEPTS (USER)' exist or can be derived from existing data, where 'EMPS' contain all employee identifiers and 'DEPTS' contain all department identifiers (assume that 'USER' in both cases is **char** 8). Then the natural join of 'CARS2' with 'EMPS' and with 'DEPTS' would isolate the 'CARS2' tuples where users are individual employees and departments respectively. This suggests the following solution:

```
u
  alt
      lnj CARS2, EMPS;
    drop USER
    add
       USERTYPE char 4 = "EMP",
       EMPNO char 3 = USER,
       DEPTNM char 8
  ;,
  alt
      lnj CARS2, DEPTS;
    drop USER
    add
       USERTYPE char 4 = "DEPT",
       EMPNO char 3,
       DEPTNM char 8 = USER
  ;
;
```

4.2.1.3 Value Scale Conflict Resolution

A value scale conflict involving a given property can be resolved with the **alt** operator by replacing or augmenting the corresponding attribute in one or both of the conflicting relations with a new attribute whose values are the original attribute's values transformed to a new value scale. Consider the following relations exhibiting a value scale conflict:

MT (CITY	MEANTEMP)
Montreal	6
Winnipeg	5
Vancouver	15

CL (CITY	CLIMATE)
Boston	cold
Norfolk	moderate
Miami	hot

The attributes 'MEANTEMP' and 'CLIMATE' represent the same property, but their value scales have different units and precision. The 'MT' relation can be transformed into another relation which is union compatible with 'CL' with the following operation:

```
alt MT drop MEANTEMP add CLIMATE char 4 =
  "cold" if MEANTEMP < 9 else
  "moderate" if MEANTEMP >= 9 and MEANTEMP < 16 else
  "warm" if MEANTEMP >= 16 and MEANTEMP < 23 else
  "hot";
```

This operation is based on the assumptions that the 'CITY' attribute is defined on the same domain in both relations, the 'CLIMATE' attribute is defined on a **char** 4 domain in 'CL', the CLIMATE' attribute value scale is the least precise of the two, and there is previous knowledge of the required evaluation function.

In general, values on a more precise scale can be directly transformed into values on a less precise scale, but the reverse is not true. In the above example it would not have been possible to transform 'CLIMATE' values to exact 'MEANTEMP' values. Note that in the transformation used in the example, information is actually lost. All information may have been retained if the 'MEANTEMP' attribute was kept in the new relation and a blank 'MEANTEMP' attribute was added to 'CL' (using **alt**). However, this could lead to confusing query results if a default value such as 0 is used for a blank 'MEANTEMP' attribute instead of a distinct 'null' value because the default value could be interpreted as a valid 'MEANTEMP' value.

In many examples of value scale conflict where the separate value scales have the same precision (or practically so) such as U.S. dollars versus Canadian dollars or grams versus ounces (where both are represented as real

numbers) loss of precision on a value scale transformation is not important and, in fact, a solution may be achieved by doing the transformation both ways so that both value scales are represented.

An alternative to transforming one conflicting attribute's values to the value scale of the other would be to transform both attributes' values to a separate, common value scale. The following example (adapted from [SMIT 81]), which refers to the 'MT' and 'CL' relations from the previous example and assumes the addition of auxiliary data in the form of relations 'AUX1' and 'AUX2' illustrates this approach:

```
AUX1 ( CLIMATE   LOWTEMP   HIGHTEMP   RANGEPROB )
       ...       ...       ...       ...
       cold      -15         0       0.35
       cold       0         15       0.25
       cold       15        30       0.15
       ...       ...       ...       ...

AUX2 ( MEANTEMP   LOWTEMP   HIGHTEMP   RANGEPROB )
       ...        ...       ...        ...
         5        -15         0        0.12
         5          0        15        0.35
         5         15        30        0.15
       ...        ...       ...        ...
```

```
RP1 == alt
           lnj MT, AUX1;
         drop CLIMATE
       ;

RP2 == alt
           lnj CL, AUX2;
         drop MEANTEMP
       ;
```

In the above example, the common scale consists of a temperature range and the probability that it occurs in a

given year in a city having the indicated 'CLIMATE' or 'MEANTEMP' value. The relations 'RP1' and 'RP2' are mergeable.

4.2.1.4 Implied Data Conflict Resolution

Many implied data conflicts can be simply solved by using alt to add an attribute to each conflicting relation and to evaluate that attribute to a different context-related constant value for each relation. For example, consider two relations 'MTLRES' and 'TORRES' corresponding to the restaurant example in Section 2.2.2.7 (they represent restaurants in Montreal and Toronto respectively, each has the same attributes, and neither has an attribute to indicate city location since this information is taken for granted when the relations are used separately in the corresponding cities). The conflict may be solved with the following operations:

```
alt MTLRES add key CITY char 8 = "Montreal";
alt TORRES add key CITY char 8 = "Toronto";
```

Note that the new attributes are added to the primary keys of their respective relations. This recognizes the fact the same tuple may exist in 'MTLRES' and 'TORRES' and represent a different restaurant in each case.

Implied information conflicts become more complex when the meaning of attributes depends on context. Consider the following relations representing parts of hypothetical

playing schedules for the Montreal Canadiens (HABS) and Toronto Maple Leafs (LEAFS) hockey clubs (example adapted from [DEEN 87]):

HABS (DATE OPPONENTS VENUE)

Jan10	Rangers	NewYork
Jan14	Flyers	Philadelphia
Jan17	Leafs	Montreal
Jan21	Jets	Montreal
Jan25	Nordiques	Quebec

LEAFS (DATE OPPONENTS HOMEAWAY)

Jan10	Oilers	H
Jan13	Whalers	H
Jan17	Canadiens	A
Jan20	Kings	A
Jan24	Canucks	A

For each relation, the name of one participating team in each game is known implicitly to local users. In the 'HABS' relation 'OPPONENTS' means "Opponents of the Montreal Canadiens", while in the 'LEAFS' relation it means "Opponents of the Toronto Maple Leafs." Suppose that an integrated schedule relation with the following schema is required:

SCHEDULE (DATE HOMETEAM AWAYTEAM)

The strategy would be to resolve conflicts by creating relations of the same schema from the "HABS" and the "LEAFS" relations, and then to merge by forming the union of the new relations. The conflict resolution and merging can be specified by the following nested operations:

```
u
  alt HABS
    drop VENUE, OPPONENTS
    add
      key HOMETEAM char 10 =
        "Canadiens" if VENUE = "Montreal" else OPPONENTS,
      key AWAYTEAM char 10 =
        OPPONENTS if VENUE = "Montreal" else "Canadiens"
  ;,
  alt LEAFS
    drop HOMEAWAY, OPPONENTS
    add
      key HOMETEAM char 10 =
        "Leafs" if HOMEAWAY = "H" else OPPONENTS,
      key AWAYTEAM char 10 =
        OPPONENTS if HOMEAWAY = "H" else "Leafs"
  ;
;
```

The derivation of the 'HOMETEAM' and 'AWAYTEAM' attributes is different for the 'HABS' and the 'LEAFS' relations, but in each case it depends on an awareness of the context of the relation. In the case of the 'HABS' relation, the knowledge that the Canadiens are the home team for any game played in Montreal is required. In the case of the 'LEAFS' relation, there is an attribute that directly indicates whether or not the Leafs are the home team. Even so, the meaning of 'HOMEAWAY' as "game in Toronto where the Leafs are the home team, or game elsewhere where the opponents are the home team" must be understood. If the 'HABS' relation had a 'HOMEAWAY' attribute, its meaning would be different due to the different context.

4.2.1.5 Derived-data Conflict Resolution

There are two fundamental ways to handle a derived data conflict between two relations A and B, where relation A has

an attribute containing values derived from other data and relation B lacks this attribute: 1) drop the derived data attribute from relation A, and 2) add a similar derived data attribute to relation B and compute values for it with an evaluation function. Note that in the second approach, if relation B does not contain the required base data for deriving the new attribute values, then it would have to be combined with one or more other relations before applying alt.

The first approach is the simplest and does not actually result in a loss of data since the value for the attribute being dropped is derived. However, for a user to see this data, the derivation would either have to be expressed in the query or in the integrated-to-external view mapping. The second approach would provide a more expressive integrated schema, but the GQML may not support the necessary functions for deriving the data.

4.2.1.6 Missing Data Conflict Resolution

The following schemata illustrate a missing data conflict:

```
from database 1:
     EMP1 ( EMPNO   EMPNAME   DEPT   OPHONE )

from database 2:
     EMP2 ( EMPNO   EMPNAME   DEPT   OPHONE   HPHONE )
```

In this example, home phone number data which is included in database 2 (attribute 'HPHONE') is simply excluded from

database 1. A missing data conflict actually exists from the database 2 user's viewpoint. If the 'HPHONE' attribute were dropped from 'EMP2' to create a relation that was then merged with 'EMP1', the resulting integrated schema would not appear to be missing any data from a database 1 user's viewpoint.

The obvious alternative resolution technique would be to add to the system a relation with attributes {EMPNO, HPHONE} that records the home phone numbers of employees represented by 'EMP1'. Then the natural join of 'EMP1' and the new relation could be merged with 'EMP2', and the integrated schema would not appear to be missing data from the database 2 user's point of view.

If 'EMP1' and 'EMP2' overlap in terms of the employees that they represent, then the auxiliary data relation would only have to record home phone numbers for employees represented by 'EMP1' only and an outer natural join would be used to create the mergeable relation. However, in this case, the relation merging technique would have to specify that 'HPHONE' values are to be drawn from 'EMP2' for the intersecting tuples.

4.2.2 Fragmentation Redefinition

This type of transform involves the redistribution of attributes and tuples among multiple relations and is used to resolve the generalization and aggregation forms of

abstraction conflict, data clustering conflicts and relationship conflicts.

4.2.2.1 Generalization Conflict Resolution

In the relational model, a generalization conflict occurs when one relation represents objects at a higher level of generalization than similarly classed objects represented by one or more other relations. A generalization conflict exists among the following global schema relations, assuming that 'EMP1' represents employees of all kinds and the other relations represent employees segregated according to job type:

```
from database 1:
     EMP1 ( EMPNO    ADDRESS    SALARY )

from database 2:
     MANAGER2    ( EMPNO    ADDRESS    SALARY )
     TECHNICIAN2 ( EMPNO    ADDRESS    SALARY )
     SECRETARY2  ( EMPNO    ADDRESS    SALARY )
```

Actually, the conflict only exists within the context of a requirement to create an 'EMPLOYEE' relation representing the entire population of employee objects in both databases, or a requirement to create specialized employee relations such as 'MANAGER', 'TECHNICIAN' and 'SECRETARY' that represent the entire population of the corresponding subclass objects from both databases. If any one of the database 2 relations are merged with 'EMP1' the result would not represent the entire population of employees. Also, if 'EMP1' represents employees from more

than one job type then the contents of the result would not be limited to employees of the job type corresponding to the database 2 relation.

One approach to resolving this conflict would be to merge the three database 2 relations into one relation, say 'EMP2', with two **u** operations. The 'EMP2' relation could then be merged with 'EMP1' to create an integrated 'EMPLOYEE' relation. This may be satisfactory from the viewpoint of the database 1 user who does not expect to see job-type data, but from the database 2 user's viewpoint there would appear to be a loss of job-type data.

An alternative approach, which would yield a more satisfactory result from the database 2 user's viewpoint, would be to divide the 'EMP1' relation into multiple relations each of which represents employees of a different job type. However, this would require auxiliary data, assuming that database 1 does not contain job type information. If this auxiliary data takes the form of a relation with the schema 'EMPJOB (EMPNO JOBTYPE)' then, for example, a relation of database 1 manager employees could be defined as follows:

```
MANAGER1 == lnj
              EMP1,
              lim EMPJOB where JOBTYPE = "MANAGER";
            ;
```

The 'MANAGER1', 'TECHNICIAN1', and 'SECRETARY1' relations defined in this manner could then be merged with 'MANAGER2',

'TECHNICIAN2', and 'SECRETARY2'. Note that if 'EMP1' represents job types in addition to manager, technician, and secretary the corresponding subclass relation (e.g. 'ENGINEER') would not be merged with a database 2 relation and so would automatically be an integrated schema relation. The database 2 user would regard this as an addition to database 2.

The objects involved in a generalization conflict may be modelled as attributes (or properties). Consider the following relations (example adapted from [DAYA 84]):

```
from database 1:
     EMP1 ( EMPNO    NAME )
     PHONE1 ( PHONENO    EMPNO )

from database 2:
     EMP2 ( EMPNO    NAME    HPHONE    OPHONE )
```

In database 2, specialized phone numbers are used (home and office numbers) while in database 1 no distinction is made between different types of phone numbers.

The relations 'EMP1' and 'EMP2' could be made union compatible and, hence, mergeable by padding 'EMP1' with blank (i.e. null or system default valued) 'HPHONE' and 'OPHONE' attributes. However, this might not be sound because one cannot automatically conclude that home and office phone numbers do not apply to, or are not known for, the employees represented in database 1. Further knowledge that the phone numbers in 'PHONE1' are neither home nor office numbers would be required to draw this conclusion.

The basic approaches to resolving the generalization conflict in this example parallels those of the previous example: depending on the desired integrated schema, the phone data in database 2 could be made more general or the phone data in database 1 could be made more specific (assuming that the required auxiliary data is available). The distinction between properties and objects having properties is only of consequence in the details of the conflict resolution operations.

The phone data in database 2 could be put into a more general form with the following operations:

```
PHONE2 == u
            ren
                lim EMP2 attrs EMPNO, HPHONE;
              attrs HPHONE to PHONE;,
            ren
                lim EMP2 attrs EMPNO, OPHONE;
              attrs OPHONE to PHONE;
          ;

EMP2b == lim EMP2 attrs EMPNO, NAME;
```

The 'PHONE2' and 'PHONE1' relations could be merged into an integrated 'PHONE' relation and the 'EMP1' and 'EMP2b' relations could be merged into an integrated 'EMP' relation.

Suppose that a comprehensive list of office phone numbers has been compiled into a relation with schema 'OPHONE (PHONENO)' and that in 'PHONE1' only office and home phone numbers are recorded. Then the following operations could be used to put the phone number data in database 1 into a more specific form:

```
OPHONE1 == lnj PHONE1, OPHONE;

HPHONE1 == dif PHONE1, OPHONE1;

EMP1b == onj
           onj
             EMP1,
             ren OPHONE1 attrs PHONENO to OPHONE;
           ;,
           ren HPHONE1 attrs PHONENO to HPHONE;
         ;
```

The onj (outer natural join) operator allows an 'EMP1' tuple to be included in 'EMP1b' even if the corresponding employee lacks one or both phone numbers.

4.2.2.2 Aggregation Conflict Resolution

The following relation schemata illustrate an aggregation conflict:

```
from database 1:
     EMP1 ( EMPNO   NAME   ADDRESS   OVERTM   SALARY )

from database 2:
     EMP2 ( EMPNO   NAME   ADDRESS )
     PAY2 ( EMPNO   OVERTM   SALARY )
```

Properties which are aggregated into a single employee object in the first database are distributed among two objects - employee and payroll - in the second database. The second database represents a more limited conception of what constitutes properties of an employee than does the first database but, at the same time, it recognizes an additional payroll object type.

The resolution of aggregation conflicts is straightforward and generally consist of using lim to

'split' a relation vertically into two or more other relations and using **lnj** or **onj** to join two or more relations into one. For the above example, a solution which conforms to the database 2 viewpoint would be as follows:

```
EMP1b == lim EMP1 attrs EMPNO, NAME, ADDRESS;
PAY1 == lim EMP1 attrs EMPNO, OVERTM, SALARY;
```

Relation 'EMP1b' could then be merged with 'EMP2' and 'PAY2' with 'PAY1'.

A solution which conforms to the database 1 point of view would be as follows:

```
EMP2b == onj EMP2, PAY2;
```

Relation 'EMP2b' could then be merged with 'EMP1'. The **onj** operation would ensure that any employees who do not have a payroll record (perhaps new employees) would still be recorded in 'EMP2b' (it would also retain payroll data where there is personnel data).

4.2.2.3 Data Clustering Conflict Resolution

The basic difference between data clustering conflicts and generalization and aggregation conflicts is that the former occur because of different performance requirements (and optimizing capabilities) at different sites, and the latter occur because of different approaches to modelling the same reality. However, data clustering conflicts have the same physical appearance as generalization and aggregation conflicts so the same basic techniques can be

applied in resolving them.

4.2.2.4 Relationship Conflict Resolution

The capability of the relational model to portray relationships with different connectivities between the same two entities can give rise to relationship conflicts. The following schemata illustrate a relationship conflict:

```
database 1:
     EMP1 ( EMPID   EMPNAME   SALARY   DEPTID )
     DEPT1 ( DEPTID   DEPTNAME )

database 2:
     EMP2 ( EMPID   EMPNAME   SALARY )
     DEPT2 ( DEPTID   DEPTNAME )
     WORKS2 ( EMPID   DEPTID )
```

Assume that all of the above relations are in Boyce-Codd Normal Form (BCNF), that 'EMPID' is the only candidate key of 'EMP1' and 'EMP2', that 'DEPTID' is the only candidate key of 'DEPT1' and 'DEPT2', and that {EMPID, DEPTID} is the candidate key of 'WORKS2'. If we interpret the 'EMPx' relations as representing employee entities and the 'DEPTx' relations as representing department entities, then it is clear that database 1 implies a many-to-one relationship between employees and departments (many employees can belong to the same department but any employee can belong to at most one department), while database 2 implies a many-to-many relationship (many employees can belong to the same department and any employee can belong to many departments).

If a single relationship is to exist between employees and departments in the integrated schema, it must be many-

to-many since only such a relationship can include all relationship instances from both databases. A solution consists of transforming the database 1 schema into a form similar to the database 2 schema since the latter already accommodates a many-to-many relationship. This would be done as follows:

```
WORKS1 == lim EMP1 attrs EMPID, DEPTID;
EMP1b == alt EMP1 drop DEPTID;
```

Relation 'WORKS1' can then be merged with 'WORKS2' to create an integrated 'WORKS' relation which represents the global many-to-many relationship. Also, 'EMP1b' can be merged with 'EMP2'. Note that the **lim** operator may have been used to derive 'EMP1b', but not as conveniently as with the use of the **alt** operator since **lim** would have required all attributes except 'DEPTID' to be listed.

When the same n entity types are portrayed in two databases, where n > 2, it is possible that in one database a ternary or higher order relationship may be portrayed while in the other database two or more binary relationships may be portrayed between the same entity types. The higher degree relationship would be used to more closely control the simultaneous relationship of three or more entities. The following schemata, adapted from [TEOR 86] give an example of this:

database 1:
INSTRUCTOR1 (IID INAME OFFICE)
CLASS1 (IID SID)
STUDENT1 (SID SNAME HOMERM)
TEAM1 (SID PID)
PROJECT1 (PID PNAME)

database 2:
INSTRUCTOR2 (IID INAME OFFICE)
STUDENT2 (SID SNAME HOMERM)
PROJECT2 (PID PNAME)
GROUP2 (SID PID IID)

Assume that all these relations are in BCNF and that in all relations except 'GROUP' the 'xID' attributes are the only candidate key attributes. For 'GROUP' we will consider two alternative assumptions: 1) where {SID, PID, IID} are the attributes of the one candidate key, and 2) where {SID, PID} are the attributes of the one candidate key.

Database 1 portrays separate binary many-to-many relationships between instructors and students, and between students and projects. Database 2 portrays a single ternary relationship between instructors, students and projects. If we interpret 'GROUP' using the first assumption then for each related <student, project> pair in database there may be any number of related instructors, as in database 1. However, unlike database 1, a relationship cannot occur between students and projects unless it also includes at least one instructor (this is so because primary key attributes cannot be null). If we interpret 'GROUP' using the second assumption, then a student and a project can be related in database 2 without a coincident relationship to an instructor, as in database 1. However, unlike database 2,

at most one instructor can be related to a related <student, project> pair.

With either assumption, database 2 has more restrictions on how the three entity types are simultaneously related to each other. Therefore, a solution to the conflict consists of transforming the database 2 schema into a form similar to that of the database 1 schema, with 2 separate binary relations. This would be done as follows:

CLASS2 == **lim** GROUP **attrs** IID, SID;

TEAM2 == **lim** GROUP **attrs** SID, PID;

Relation 'CLASS2' would be merged with 'CLASS1', and 'TEAM2' with 'TEAM1' so that in the integrated schema the less restrictive combination of two binary relations would be used.

4.2.3 Transposition

This kind of transform is based on the application of the **trc** and **tcr** operations to replace attribute names by attribute values and vice versa. Transposition transformations are used to resolve row/column conflicts. An example of a transposition transformation which resolved a simple row/column conflict was given in Section 3.1.6. A more complex and general example of a row/column conflict is exhibited by the following relations (example adapted from [KENT 82]):

from database 1:

EMP1 (EMPNO EMPNAME SALARY)

EMPNO	EMPNAME	SALARY
123	John	25
124	Mary	30
125	Greg	27

from database 2:

EMP2 (EMPNO FACT TYPE VALUE)

EMPNO	FACT	TYPE	VALUE
123	NAME	STRING	John
123	SALARY	INTEGER	25
124	NAME	STRING	Mary
124	SALARY	INTEGER	30
125	NAME	STRING	Greg
125	SALARY	INTEGER	27

Relation 'EMP2' simulates a binary relational approach to modelling employee data. Though perhaps slightly far-fetched, this schema might be used in order to facilitate the addition of new employee properties. The 'VALUE' attribute is defined on a character string domain with a large width so that it would likely accommodate character string representations of applicable property values (say **char** 30). The 'FACT' and 'TYPE' attributes provide the information required to correctly use the property values. Relation 'EMP1' has a more conventional schema where the 'EMPNAME' and 'SALARY' attributes are defined on different domains (say **char** 10 and **numeric** 3 respectively). Although the 'EMP1' schema is clearer than that of 'EMP2', it does not accommodate the addition of new properties.

The 'EMP1' relation may be transformed into a form similar to that of 'EMP2', thus enabling a integrated schema that is preferable from the database 2 user's viewpoint, by the following operations:

```
EMP1b == alt EMP1
           drop EMPNAME, SALARY
           add
             EMPNAME char 30 = EMPNAME,
             SALARY char 30 = numeric_to_char (SALARY)
         ;

EMP1c == trc EMP1b row EMPNAME, SALARY
                   to_col VALUE seq SEQ;

EMP1d == alt EMP1c
           drop SEQ
           add
             TYPE char 10 = "STRING" if SEQ = 1
                            else "NUMERIC,
             FACT char 10 = "EMPNAME" if SEQ = 1
                            else "SALARY"
         ;
```

Assuming that the 'TYPE' and 'FACT' attributes in 'EMP2' are defined on **'char** 10' domains, 'EMP1d' is mergeable with 'EMP2'. The first operation to define 'EMP1b' from 'EMP1' replaces 'EMPNAME' and 'SALARY' with new attributes defined on a common domain so that both of the new attributes can be specified in the same **'row'** clause. Furthermore, the common domain is that of the 'VALUE' attribute in 'EMP2'. The second operation to define 'EMP1c' from 'EMP1b' is the actual transposition in which 'EMPNAME' and 'SALARY' values are combined in a single 'VALUE' attribute. Because of the order of specification in the **'row'** clause, values originating from 'EMPNAME' are related to a 'SEQ' value of 1, and those originating from 'SALARY' are related to a 'SEQ' value of 2. The final operation to define 'EMP1d' from 'EMP1c' converts the 'SEQ' values to corresponding 'FACT' and 'TYPE' values.

The reverse transformation to put 'EMP2' into a form

similar to that of 'EMP1' would be specified as follows:

```
EMP2b ::= alt EMP2
            drop FACT, TYPE
            add SEQ numeric 1 = 1 if FACT = "EMPNAME"
                      else 2
          ;

EMP2c ::= tcr EMP2b col VALUE to_row EMPNAME, SALARY;

EMP2d ::= alt EMP2c
            drop EMPNAME, SALARY
            add EMPNAME char 10 = EMPNAME,
                SALARY numeric 3 =
                          char_to_numeric(SALARY)
          ;
```

Note that for this mapping to remain valid, tuples with a new 'TYPE' attribute cannot be added to 'EMP2'. This would impose a possibly unacceptable global constraint on database 2.

4.2.4 Tuple Grouping

This type of transform is based on the application of the **grp** operator and is used to resolve the set abstraction and summarization of properties forms of abstraction conflict.

4.2.4.1 Set Abstraction Conflict Resolution

The following schemata illustrate a set abstraction conflict (example adapted from [DAYA 84]):

```
from database 1:
     SHIP1 ( SHIPID   WEIGHT   LOCATION   CAPTAIN )
     SHIPCONV1 ( SHIPID   CONVOYID )

from database 2:
     CONVOY2 ( CONVOYID, AVGWT, LOCATION )
```

Individual ships are represented in database 1 while a higher-level set abstraction of ships, the convoy, is represented in database 2. In order to devise a strategy to handle the set abstraction conflict, it is necessary to know which convoys the individual ships in 'SHIPS1' belong to. In the example, this data resides in the existing relation 'SHIPCONV1', but it could also be auxiliary data. The following derivation is then possible:

```
SHIP1b == lnj SHIP1, SHIPCONV1;

CONVOY1 == grp SHIP1b by CONVOYID, LOCATION
             add AVGWT numeric 3 = avg(WEIGHT)
           ;
```

This transformation assumes that 'SHIP1' represents all ships in a given convoy. Relation 'CONVOY1' is mergeable with 'CONVOY2' (assume that 'LOCATION' is functionally dependent on 'CONVOYID' in 'CONVOY1' and 'CONVOY2'). Also, 'SHIP1b' may be retained in the integrated schema so that both detailed and abstracted ship data is available. Note, however, that if 'CONVOY1' and 'CONVOY2' are not disjoint and there is a conflict in the 'AVGWT' values of intersecting tuples then the individual ship data for the corresponding convoys is usable only if the intersecting 'CONVOY2' tuples are used and the intersecting 'CONVOY1' tuples are rejected.

If 'CONVOY2' represents ships that are not represented by 'SHIP1' then in order to conform to the database 1 user's viewpoint it would be necessary to convert convoy data to

individual ship data. However, this would essentially entail augmenting the database with further individual ship data.

4.2.4.2 Summarization of Properties Conflict Resolution

For an example of a summarization of properties conflict, consider the following relation schemata:

from database 1:
EMP1 (EMPNO EMPNAME DEPTNAME SALARY)

from database 2:
EMP2 (EMPNO EMPNAME DEPTNAME)
DEPTSAL2 (DEPTNAME AVGSALARY)

In database 1 the salary of each individual employee is recorded while in database 2 only the average salary of all employees in each department is recorded (the natural join of 'EMP2' and 'DEPTSAL2' would create a relation where average departmental salary is directly modelled as a property of each employee). One conflict resolution approach would be as follows:

```
EMP1b == alt EMP1 drop SALARY;

EMPSAL1 == lim EMP1 attrs EMPNO, SALARY;

DEPTSAL1 == grp EMP1 by DEPTNAME
              add AVGSALARY numeric 6 = agg_avg(SALARY)
            ;
```

This transformation assumes that 'EMP1' represents all employees in the departments whose names are recorded in 'EMP1'. The definition of 'EMP1b' and 'EMPSAL1' resolves a data clustering conflict while avoiding the loss of detailed salary data ('EMP1b' may be merged with 'EMP2' and 'EMPSAL1 would be included in the integrated schema). Relation

'DEPTSAL1' may be merged with 'DEPTSAL2'. In the event of a data conflict between 'DEPSAL1' and 'DEPSAL2', 'EMPSAL1' would only be usable if the intersecting 'DEPSAL1' tuples are used and the intersecting 'DEPSAL2' tuples are rejected.

4.2.5 True/False Transform

This type of transform, described in [KENT 82], consists of changing a relation in which the truth of a fact is implied by the presence of corresponding tuple in that relation to one where both truths and falsehoods are represented by tuples (which are marked accordingly), and vice-versa. The true/false transform is used to resolve implicit/explicit truth conflicts.

The following relations provide an example of an implicit/explicit truth conflict:

from database 1:

FLDACCS1

TEAM	FIELD	TRUTH
A	North	TRUE
A	South	TRUE
B	North	FALSE
B	South	TRUE
C	North	TRUE
C	South	FALSE

from database 2:

FLDACCS2

TEAM	FIELD
D	North

Relation 'FLDACCS1' considers teams 'A', 'B', and 'C', fields 'North' and 'South' and shows which team has access to which field by listing all combinations of teams and fields and indicating for each combination whether it is

true or false that the team has access to the field. Relation 'FLDACCS2' shows that team 'D' has access to the 'North' field (the truth of this fact is implicit).

The above conflict may be resolved from the database 2 user's viewpoint with this simple operation:

```
FLDACCS1b == lim FLDACCS1 where TRUTH = TRUE
                          attrs TEAM, FIELD;
```

Relation 'FLDACCS1b' may then be merged with 'FLDACCS2'.

The opposite transformation for resolving the conflict from the database 1 user's point of view is not so simple. One approach consists of the following steps. First, all the applicable 'TEAM' and 'FIELD' values for relation 'FLDACCS2' are be identified (these values are actually the domains of 'TEAM' and 'FIELD', but the if the implemented domains are specified as character strings of certain lengths, for example, then the actual domains cannot be deduced from the specifications). Secondly, an auxiliary data relation must be created which is the cartesian product of the 'TEAM', 'FIELD' and truth values with a FALSE 'TRUTH' value appended to each tuple, for example:

```
AUX ( TEAM   FIELD   TRUTH )
      D      North   FALSE
      D      South   FALSE
      E      North   FALSE
      E      South   FALSE
      ...     ...     ...
```

Essentially this indicates field access, with respect to database 2, in the case where 'FLDACSS2' is an empty

relation.

Thirdly, the auxiliary data is combined with 'FLDACCS2' so that facts portrayed by 'FLDACCS2' have precedence. This may be done with the following operations:

```
AUXb == dif
          AUX,
          lnj FLDACCS2, AUX;
        ;

FLDACCS2b == u
               alt FLDACCS2 add TRUTH = TRUE;,
               AUXb
             ;
```

The first operation defines relation 'AUXb' which contains tuples from 'AUX' that do not contradict the facts in 'FLDACCS2'. The second operation defines relation 'FLDACCS2b' which is union compatible with 'AUXb' and 'FLDACCS1' and that contains the facts from 'AUXb' and 'FLDACCS2'. Relation 'FLDACCS2b' can be merged with 'FLDACCS1'.

Note that in the above example, we assume that the domains of the 'TEAM', 'TRUTH', and 'FIELD' attributes are the same in all relations, and that the 'TEAM' and 'FIELD' attribute domains are specified as fixed length character strings. Furthermore, we assume that in both databases, only "North" and "South" fields are relevant. If database 2 had included 'FIELD' values other than these, then it would have been necessary to include a 'negative fact' tuple for each database 1 team and each such 'FIELD' value in 'AUX'.

CHAPTER 5 A QUERY PROCESSING ALGORITHM

This Chapter describes the design of a query processing algorithm used in the HDDBMS prototype implementation described in Chapter 6 is based. The algorithm processes ad hoc queries in the GQML described in Chapter 3. It is intended to be appropriate for an HDDBMS that would act as a front-end to simple, micro-computer based local DBMSs. Specific assumptions used in its design are as follows:

1) The HDDBMS sites would be connected by fast short haul links, so intersite communication delay would not dominate data processing delay.

2) Each local DBMS would be capable of supporting any GQML operation, so no local DBMS would have to compensate for limitations of another local DBMS, nor would the HDDBMS have to include any data processing capabilities of its own.

3) Local DBMSs would not maintain statistics on the sizes of their database entities which could be used to estimate the size of mapping and query operation result relations.

Our query processing algorithm decomposes a global query into a set of subqueries each of which is executed entirely at exactly one site in the HDDBMS. Multiple subqueries are executed in parallel, their results are collected at the query site, and a final subquery on the intermediate results is executed at the query site to

produce the query result.

With our algorithm, a join on relations residing at different sites would always be executed at the query site. However, schemes for optimizing these joins such as the semijoin algorithms used in SDD-1 [GOOD 81] and Mermaid [TEMP 87], and the enumeration of join sequences, join sites, and join methods as in R* [DANI 82] and Multibase [DAYA 85], require database statistics which we assume would not be available to our algorithm. Also, semijoin algorithms are usually only worthwhile when the sites are joined by slow long-haul links where communication costs dominate data processing costs. The fragment-and-replicate algorithm mentioned in Section 2.3 is designed for sites joined by a fast local area network, but it assumes either entirely replicated or disjoint global relations. This is a questionable assumption in a HDDBMS. Our algorithm places no conditions on data distribution or the degree of data overlap between different sites.

5.1 OVERALL ALGORITHM DESIGN

The overall query processing algorithm (qproc) is outlined in Figure 5.1. Each site in the HDDBMS would have its own unique site designator and its own process running this algorithm. A GQML query submitted to the qproc process running at a particular site consists of a result relation name and an GQML operation, or hierarchy of nested GQML

operations, on terminal and/or non-terminal relations specified at that site with GQML virtual relation definitions and base relation declarations. The local site is the site where the qproc process is running, and the query site is the site where the query originates from, and where the results are required.

If the query site designator is the same as the local site designator, then the query is local, otherwise it is remote. Local and remote queries are handled in exactly the same way, except that the result of a local query remains at the local site, in a file having the format used by the local DBMS, while the result of a remote query is written to a global format file which is sent to the query site.

The basic feature of qproc is that a query on virtual relations at a given site (the query site) is decomposed into zero or more subqueries that are entirely satisfied by local data (local subqueries), zero or more subqueries on virtual relations which, according to the locally stored mapping specifications, are not derived from local data (remote subqueries), and a final subquery on the results of the local and remote subqueries and/or translated local host schema relations. The query decomposition algorithm will be explained in Section 5.3. The important thing to note at this point, however, is that local and remote subqueries can be further processed and executed in parallel.

```
algorithm qproc
/* query processing */

  input:   (1) GQML query
           (2) local site id
           (3) query site id

  output:  at query site, file containing query result
           or an error indicator
  {
  decompose query into final subquery, local subqueries
  and remote subqueries;
  (algorithm decomp_qry)

  if (no errors in decomp_qry) {
    dispatch remote subqueries, if any, to remote sites
    for further processing and execution;

    transform local subqueries, if any, and submit to
    local DBMS for parallel execution with any remote
    subqueries;

    wait until results of all remote subqueries have
    been received or until at least one error indicator
    has been received;

    if (no errors in  processing remote and local
        subqueries )
      transform final subquery and submit to local DBMS
      for execution, producing, on success, a local
      format file if query is local, or a global format
      file if query is remote;
    }

  if ( query is remote )
    if (no errors in decomposition or subquery
        processing)
      transfer results to query site;
    else
      send error indicator to query site;

  if ( query is local and error encountered )
    output error indicator locally;

  erase intermediate result files;
  }
```

Figure 5.1 query processing algorithm

Local and final subqueries are translated into a form that can be executed by the local DBMS. When a translated subquery is submitted to the local DBMS the qproc process no longer has any involvement in it except to determine whether or not the execution was successful. The translated final subquery may be augmented with instructions to the local DBMS to present the user with options for activities such as viewing the final result, and erasing the result file. If data reformatting capabilities are an integral part of the local DBMS, it might make sense to also include, in the transformed final subquery, instructions to convert received global format remote data (the results of remote subqueries) to local format, rather than to perform the reformatting immediately on receipt of the data.

If there are no remote subqueries, then there will be no local subqueries, and the entire query will be executed as the final subquery. If there are no local subqueries and only one remote subquery then the result of that remote subquery would contain the query result data. In this case, the final subquery would be 'null' and would result in the activities that would be appended to any other final subquery upon translation, such as data reformatting and results display.

When a remote subquery is dispatched to a remote site, it is submitted as an ordinary query to the qproc process at that site (i.e. the receiving qproc process does not need to

know that the query is actually a subquery at another site), which then processes it and returns the results to the dispatching qproc process site. When processing a remote query which happens to be a subquery at another site, the qproc process may itself dispatch remote subqueries if the local DBMS is, in fact, a separate HDDBMS.

The subqueries returned by decomp_qry would be described by an internal representation of a directed acyclic graph of GQML operations, which is described in Section 5.2. Such a representation would be the starting point of the transformation of a subquery into any other DBMS specific representation, or in to a representation suitable for transmission to a remote site. Subqueries are intended to be transmitted in text files as GQML text in a typical HDDBMS implementation. The translation from the internal representation to GQML text, described in Section 5.7.1, would be part of the dispatching function.

It may be desirable for one site to send multiple subqueries to another site as a group in order to minimize communication setup cost and delay. It would then also be desirable for the results of each subquery in such a group to be returned as a group. In this case, there would be an intermediate module to individually submit each subquery received in a group to the query processor, and then collect the results for transfer after the final result file is created.

5.2 DAG REPRESENTATION OF A QUERY

If we replace the non-terminal operands of a GQML query with their GQML definitions, the result is essentially the same query (terminal operands are defined in Section 3.5). The repetition of this substitution until all operands of the query are terminal relations modifies a GQML query on terminal and/or non-terminal relations into the same query on terminal relations only. Query modification is exemplified in Figure 5.2 which is based on the integrated schema of Figures 3.1 to 3.5. For clarity, only the operand list of each operation is shown. The terminal relations involved in the query (assuming that site 1 is the query site) are 'DEPT2', 'EMP2', 'WORKS', and 'EMP'. The first two terminal relations are global relations which correspond to site 2 local participant relations, and which map to translated local host schema relations (and, ultimately, data) at site 2. The last two terminal relations are translated local host schema relations at site 1 and, therefore, correspond to data stored at site 1.

unmodified query (submitted at site 1):

```
RESULT := lnj WORKSIN, EMPLOYEE;
```

modified query:

```
RESULT := lnj
            u
              lim EMP2 ...;,
              ren WORKS1 ...;
            ;,
            alt
              onj
                ren EMP ...;,
                alt EMP2 ...;
             ;
            ;
          ;
```

Figure 5.2 GQML query modification example

The modified query can be represented as a operation DAG by transforming each operation and each terminal relation into a node and directing edges from an operation to its operand(s). Therefore, the successor node(s) of a particular operation node (there may be either one or two) represent the operand(s) of that operation. The DAG corresponding to Figure 5.2 is illustrated in Figure 5.3. The terminal relations are indicated as '**term**(inal)' nodes and the names of operation result relations are indicated in parenthesis where such names are given in the mapping specifications and the query.

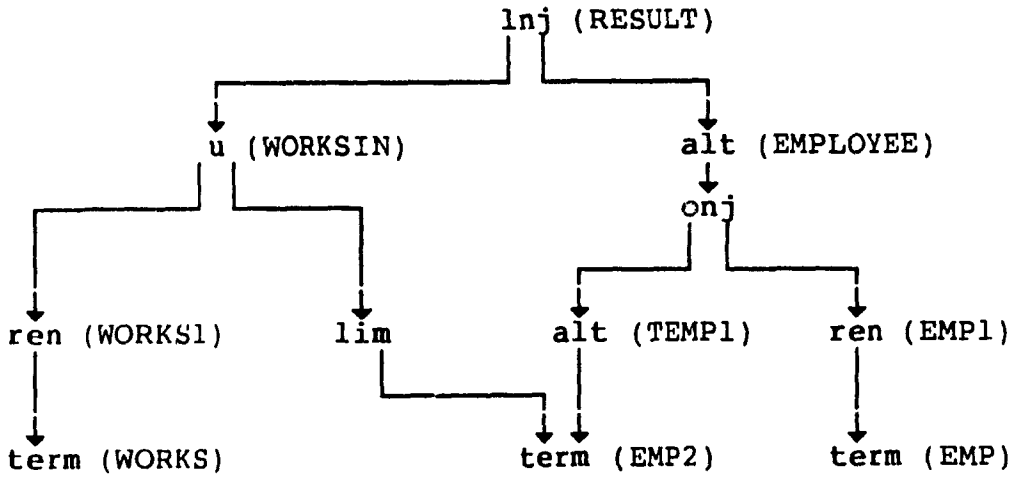


Figure 5.3 operation DAG example

In an operation DAG, terminal nodes cannot have outgoing edges. Also, there is one node with no incoming edges. This is called the result node since it represents the final operation that must be executed in order to materialize the query result relation. A DAG consisting of only a terminal node does not represent a query; at least one operation is required. The simplest operation, which returns a copy of a base relation, is **lim** without a **where** or an **attrs** clause.

In the PRECI* system, queries are also represented as graphs of extended relational algebra operations, but only trees are built [DEEN 85b]. This means that if the same virtual or terminal relation is referred to multiple times in the user query, or in the mapping operations for virtual relations referred to in the user query, then the same subtree will occur multiple times in the query operation

tree. Such duplicate subtrees would lead to duplicate processing and, possibly, duplicate subqueries, but this problem is not considered in [DEEN 85b]. Our DAG generation algorithm (Section 5.4) automatically merges multiple references to the same virtual or terminal relation into a single subgraph.

5.3 QUERY DECOMPOSITION

The query decomposition algorithm (decomp qry), outlined in Figure 5.4, is based on the decomposition of a directed acyclic graph (DAG) of operations representing the query into separate subgraphs representing the subqueries. An explanation of this algorithm follows.

In decomp_qry as well as in qproc, the query and subqueries are represented by operation DAGs, and, for convenience, the query or a specific subquery is referenced by the result node of the corresponding DAG. Algorithm decomp_qry initially obtains the operation DAG for the query using algorithm gen_DAG, discussed in Section 5.4.

In addition to creating the query operation DAG, gen_DAG assigns to each node a set of site id's, called a 'site set'. The site set of a terminal node represents all the sites where the corresponding relation is in either the local participant schema or translated local host schema. There is one exception to this: if the relation is in the local site's local participant schema then the terminal

node's site set will be assigned only the local site id. The reason for this will be explained later.

The site set of a non-terminal (or operation) node is normally the intersection of the site sets of its adjacent successor nodes, as in the PRECI* query decomposition algorithm [DEEN 85a] (adjacent successor nodes will be referred to simply as 'successor nodes' from here on). A node with a non-empty site set is classified as 'single-site' since the corresponding relation may be entirely materialized through the DBMS at any of the sites in its site set. To materialize the relation corresponding to a non-single-site operation node, DBMSs at more than one site need to be accessed. The site set of a operation node may be set to empty even if the intersection of the site sets of its successors is non-empty if it is decided (in gen_DAG) that the operation should not be executed remotely. This is discussed in more detail in Section 5.4. Algorithm gen_DAG is also required to produce a list of all the nodes in the query DAG.

A node with a site set containing only the local site designator will be referred to as 'local', any other node with a non-empty site set will be referred to as 'remote'.

```
algorithm decomp_qry
/* query decomposition */

  input:    (1) GQML query
            (2) local site id

  output:   (1) final subquery result node
            (2) set of local subquery result nodes
            (3) set of remote subquery result nodes
  {
  initialize local and remote subquery sets to NIL;
  initialize node list to NIL;
  initialize final subquery to null;

  generate query operation DAG;
  (algorithm gen_DAG)
  if (error in gen_DAG)
    return with error indicator;

  apply improvement transformations to query DAG;
  (algorithm improve_DAG)

  if (result node of query operation DAG is single
      -site)
    if (node is not local)
      add query result node to remote subquery set;

    else {
      set final subquery result node to query result
      node;

      return;
      }

  else {

    set final subquery result node to query result
    node;

    for (each node on node list)
     if (current node is not single-site)
        for (each successor node)
          if (current successor node is
              1) single-site and
              2) not marked as a subquery result node)
            {
            mark current successor node as a subquery
            result node;

            if (current successor node
                1) is terminal and
                2) is not local)
```

```
                {
                create a lim node with empty  where and
                attrs clauses, make the current successor
                node its successor node, and assign it a
                copy of the current successor node's
                site set;

                set the current successor node to the
                new lim node;
                }

              if (current successor node is local)
                add to local subquery set;
              else
                add to remote subquery set;
              }
        }

      select remote subquery sites;
      (algorithm sel_sites)
      if (error in sel_sites)
        return with error indicator;
      }
```

Figure 5.4 query decomposition algorithm

Before the actual query decomposition is performed, algorithm improve_DAG is used to transform the final subquery DAG (which at this point is also the query DAG) into an equivalent DAG which represents a more efficient query expression. By 'equivalent' we mean that if the query were to be fully processed twice, once with and once without the transformation, the two result relations would contain exactly the same data. This transformation may result in nodes being removed from and/or added to the DAG, in which case algorithm improve_DAG would change the node list accordingly. Algorithm improve_DAG has not been designed in detail, but aspects pertinent to it are discussed in Section 5.5.

If the query result node is single-site, then this means that the entire query can be executed using the DBMS of one site. If the query result node is local, then decomp_qry will make the final subquery equivalent to the query, which would cause the entire query to be executed using the local DBMS. This will mean no data transfer cost, although there will not be any parallel data processing either.

If the query result node is remote, then the entire query will be executed at a remote site, and the final subquery will remain null. Data transfer cost will be incurred for the final results only, and again there will be no parallel data processing. If the site set contains multiple site designators then some processing is required to select a single subquery site from the set. This processing, performed by algorithm sel_sites, will be discussed in Section 5.6. For the moment the query result node is simply inserted into the remote subquery set.

If the query result node is not single-site then the query can be decomposed into a collection of remote and local subqueries and a non-null final subquery. To do this, each node on the node list is inspected to determine whether it is non-single-site. If for a particular node this is true, then the successor(s) of that node are inspected. If a successor is single-site, has not already been marked as a subquery result node, then it represents either a new

subquery result node, or a terminal node in a new subquery. In the latter case, a **lim** operation node needs to be created with the terminal node as its successor (since a terminal node alone does not constitute an operation DAG) and this becomes the subquery result node. The **lim** operation would have empty **where** and **attrs** clauses, making the subquery simply a request for a copy of an entire relation. All Local and non-local (or remote) subquery result nodes are added to the local and remote subquery sets respectively. Finally a submission site is selected for each remote subquery using algorithm sel_sites.

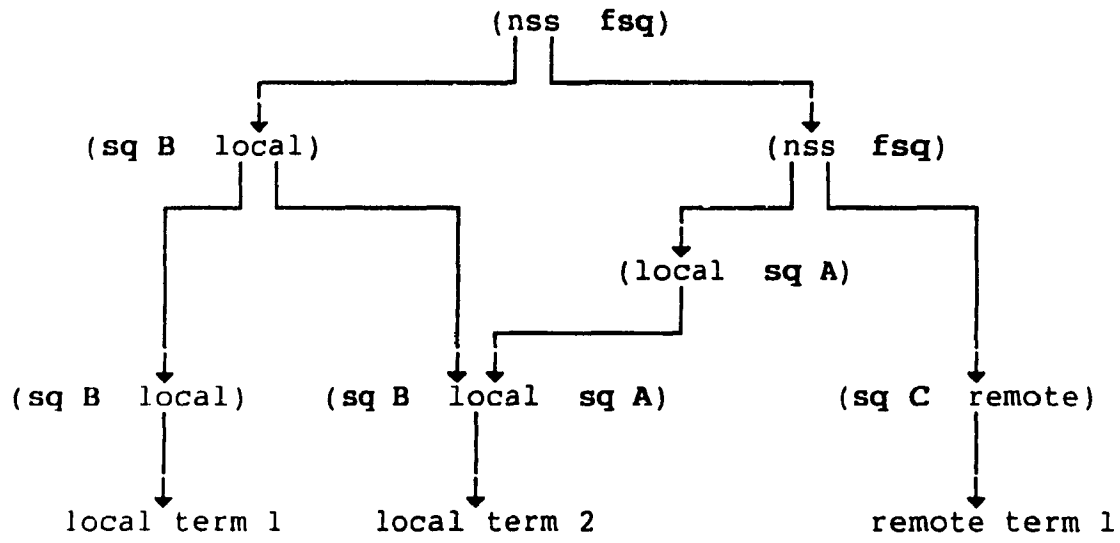
Marking a node as a subquery result node serves two purposes. First, it prevents the creation of redundant subqueries. This might otherwise occur because it is conceivable that the same single-site node can be the successor of multiple non-single-site nodes. Secondly, it marks the node as a terminal node of the final subquery.

The policy of excluding remote site designators from the site set of a local terminal node would cause decomp_DAG to favour local processing. This in turn would help minimize the transfer of data to the local site. In many cases it also has the effect of reducing parallelism since an operation that could be executed remotely or locally would always be executed locally. The extreme case of this is seen when the query node turns out to be local or remote. However, without good estimates of the relative costs of

executing specific operations on specific data at specific sites, it is not possible to determine an allocation of operations that would maximize parallelism.

It is possible that a subgraph which is part of a particular local subquery DAG may also be part of another local subquery DAG, in other words, local subquery DAGs may intersect. An example of this is illustrated in Figure 5.5; the operation node just preceding the 'local term2' node belongs to both subquery A and subquery B. The intersecting subquery DAGs would not be considered as one subquery DAG since there are two result nodes.

The intersection subgraph could be considered as a separate local subquery, but this is not done in order to avoid the extra processing needed to ensure correct subquery execution sequencing. For example, if the intersection subgraph in Figure 5.5 is interpreted as a separate local subquery, then that subquery would have to be executed before subqueries A and B. Alternatively, the result relation corresponding to the intersection subgraph could be materialized twice, once for each of the intersecting subqueries, but this is clearly wasted effort.



nss : non-single-site operation node
local : local operation node
remote : remote operation node
term : terminal node (remote or local)

fsq : final subquery node
sq X : operation node in subquery X (remote or local)

Figure 5.5 example of overlapping subqueries

What is done is that when a local subquery is transformed and executed, the intermediate result relations corresponding to each operation node (not just the result node) are not erased until after the final subquery is executed. Furthermore, the operation nodes of the DAG are marked to indicate that the corresponding result relations have been materialized. Therefore, when a local subquery whose DAG intersects with that of an already executed local subquery is processed, the translation routine can see that the result node of the intersection subgraph is now a terminal node of the current local subquery, and the DBMS can be instructed to use the corresponding intermediate

result relation as a base relation. The order of execution of intersecting subqueries clearly does not matter with this method.

For example, suppose that subquery A of Figure 5.5 is executed before subquery B. Then the operation node belonging to both subqueries (intersection operation node) would be marked in the translation of subquery A, and after the execution of subquery A the corresponding result relation would be materialized. When subquery B is translated, then the intersection operation node would be treated as a terminal node. If subquery B happened to be executed first, then the intersection subquery node would be treated as a terminal node for subquery A.

Note that it is possible for one subquery to be a subset of another. For example, consider Figure 5.5 without the 'local **sq A**' node directly above the intersection operation node. In this case the subquery A DAG would be a entirely included within the subquery B DAG, so subquery A would be a subset of subquery B. If subquery B were to be translated and executed first, then the subsequent attempt to translate and execute subquery A would simply return without doing anything. Of course, the formation of local subqueries which are subsets of other local subqueries could be avoided in the first place by checking whether a potential local subquery result node has a local predecessor node; if so then the local subquery would not be created.

However, this would require maintaining pointers from a node to all its predecessors, or examining the successors of many nodes each time a new local subquery is considered.

Remote subquery DAGs may also intersect. However, the global query processing does not ensure that the relations corresponding to the operation nodes of the intersection DAG are materialized only once. This, in fact, is not necessarily desirable because parallelism may be increased and overall delay decreased by the assignment of remote subqueries having intersecting DAGs to different sites, even though some intermediate results would be computed more than once.

Remote and local subquery DAGs cannot possibly intersect since it is impossible for any node in a remote subquery DAG to contain the local site id in its site set, and all nodes in a local subquery DAG contain only the local site id in their site sets.

In the case where multiple remote subqueries are received as a group from the same site, it would be useful to ensure that common intermediate results are computed only once. However, this would require changes to the current algorithms which currently generate subqueries independently of one another and consider each incoming subquery individually. These changes constitute a topic for further work (see Chapter 7).

5.4 DAG GENERATION

This Section describes an algorithm for constructing the query operation DAG directly from the query text and the mapping text at the local site. This algorithm is based on recursive descent parsing involving mutual recursion between algorithms gen_DAG (Figure 5.6) and gen_from_text (Figure 5.7), where gen_DAG is initially called from decomp qry.

Algorithm gen_DAG returns with an error indicator if it sees that a cycle will be introduced in the operator graph. A cycle would correspond to a modified query with an operand having the same name as the result relation name. The cycle check is made possible through the use of a node list. Algorithm gen_DAG puts a 'placeholder' node on the node list containing the result relation name before the subgraph corresponding to that result relation is actually constructed. If a subsequent recursive execution of gen DAG finds a placeholder node on the global node list which matches its result relation name input, then a cycle is discovered (recall that the result relation name is part of the query input). When the subgraph is constructed without a cycle being discovered, the placeholder node is replaced by the result node. If gen_DAG finds a non-placeholder node on the global node list which matches its result relation name input, then the algorithm simply returns with that node. Such a node represents a virtual or a base relation which is an operand in multiple GQML expressions. At the end of DAG

construction, every operator node will be on the node list.

```
algorithm gen_DAG
/* generate DAG corresponding to a GQML query */

  input:   (1) GQML query (result relation name and
              query text)
           (2) local site id
           (3) node list

  output:  result node of directed acyclic graph (DAG)
           of operations representing the input query

  {
  search node list for a listed_node corresponding to
  result relation name, return NIL if node not found;

  if (listed_node not NIL)
    if (listed_node is a placeholder)
      /* result relation_name has been referred to in
         another subquery, but the corresponding
         subgraph has not been constructed
      */
      return with error indicator;
    else
      return with listed node;
  else {
    add placeholder node containing result relation
    name to node list;

    using operation text generate result node (and
    associated DAG);
    (algorithm gen_from_text)

    if (error in gen_from_text)
      return with error indicator;

    replace placeholder node by result_node;

    return with result node;
    }
  }
```

Figure 5.6 algorithm for generating query operation DAG

When gen_DAG successfully completes the cycle test it calls gen_from_text to actually process the GQML text, which could either be an operation (or hierarchy of nested

operations) or a base relation declaration. Algorithm gen_from_text recursively parses the GQML text and converts it to the internal operator DAG representation. When the successor nodes of the current node have been created, gen_from_text adds information to the current node required for subsequent query processing while checking the validity of the text (this would involve separate parsing routines). This information includes the schema of the corresponding relation. In the case of an operation, this is the result relation schema and it is inferred from the operation and the relation schemata associated with the successor nodes (i.e. the schemata of the operation result relations). In the case of a base relation declaration, the schema is directly given by the operation text. Algorithm gen_from_text generates its own unique result relation name for the current node if the input result relation name is null. This would be the case for a recursive call from gen_from_text. A further algorithm (comp_site_set) describes the computation of the site set of the current node.

When the GQML text input is a relation name rather than operation or base relation declaration text, gen_from_text gets the GQML mapping text corresponding to the relation name (either operation or base relation declaration text) and creates a query from it, which it submits to gen_DAG along with the node list. If the GQML text input is a base relation declaration, gen_from_text does not need to call either itself or gen_DAG.

Algorithm comp_site_set (Figure 5.8) is straightforward except for two aspects: 1) obtaining the site id's of all sites containing a particular relation in their local participant schema, and 2) determining permissible remote operations.

The most direct way of handling the site id problem would be to keep lists at each site which hold the names of the relations in each of the other sites' local participant schema. Algorithm comp_site_set would then simply refer to these lists. This would be practical as long as additions and deletions to local participant schemata are infrequent, because each such change would mean a change to a relation name list at one or more other sites.

```
algorithm gen_from_text
/* generate DAG corresponding to GQML text */

  input:   (1) GQML text (operation, base relation
               declaration, or relation name)
           (2) local site id
           (3) node list
           (4) result relation name (null if call is
               recursive)

  output:  result node of non-decomposed DAG
           corresponding to the operation text

  {
  if (operation text represents a terminal relation) {
    create corresponding terminal node;

    compute site set of terminal node;
    (algorithm comp_site_set)

    return with terminal node;
    }

  else {
    create an empty operation node;

    for (each operand indicated in text) {
      if (current operand is an operation) {
        generate operand result node;
        (algorithm gen_from_text)
        if (error in gen_from_text)
          return with error indicator;

        put operand result node on global node list;
        }
      else /* operand is a relation name */
        get corresponding mapping text;

        if (corresponding mapping text does not exist)
          return with error indicator;

        generate operand result node;
        (algorithm gen_DAG)
        if (error in gen_DAG)
          return with error indicator;

      create edge from operation node to
      operand result node;
      }

    fill in operation node details, referring to
    successor nodes to determine validity of
    operation specifications and infer result relation
    schema;
```

```
        if (error found while filling in operation node)
          return with error indicator;

        compute site set of operation node
        (algorithm comp_site_set)
        if (error in comp_site_set)
          return with error indicator;
        }

      return with operation node;
      }
```

Figure 5.7 algorithm for generating DAG from operation text

```
comp_site_set
/* compute the site set of a DAG node */

  input:  (1) DAG node
          (2) local site id;

  output: DAG node with site set evaluated

  {
  initialize site set to NIL;

  if (node is terminal)
    if (node corresponds to a relation in local
        site's translated local host schema) {
      add local site id to site set;
      return with node;
      }
    else {
      get site id's of all sites containing
      corresponding relation in their local participant
      schema and add to site set;

      if (no such sites)
        return with error indicator;

  else

    if (node represents a permissible remote operation)
      compute intersection of successor nodes' site
      sets and add these sites to site set;
  }
```

Figure 5.8 algorithm to compute site set of a DAG node

The alternative to maintaining relation name lists is to acquire this information from the remote sites during query processing. At the outset of processing a query on the integrated schema (a global query), the site id's corresponding to each global schema relation that does not map to a local participant schema relation (at the query site) could be compiled all at once. This would mean that each remote site would be asked for information only once, but one or more remote sites may be unnecessarily asked since not all global relations are necessarily involved in a given global query. In this case, unnecessary delay could be incurred if such a remote site is particularly slow in responding. Note that if the query is on the local participant schema then remote site information would not be required at all (this would be the case if the query is itself a subquery from a remote site).

This approach of acquiring the location information at the outset of query processing could be improved if the global schema relations associated with each integrated schema relation (through the integration mapping) were indicated. Then, the global schema relations relevant to a particular global query could be readily determined, and site information for only those relations would be sought.

Whether an operation should be a permissible remote operation depends on the effect that operation would have on the volume of data to be transferred from the remote site to

the query site. In many cases, if a remote subquery is made to contain as many operations as possible, the volume of data that will have to be transferred from the remote site to the query site will be minimized, which is clearly desirable. This would certainly be the case if the remote subquery consists of **u**, **int**, **dif**, **div**, and **lim** operations only. However, some operations, if performed remotely, may increase the volume of data to be transferred. For example, the cartesian product of two relations , performed remotely, will certainly result in a greater amount of data transfer compared to transferring the original relations and performing the cartesian product at the query site (this assumes that at least one operand relation contains more than one tuple). A cartesian product is equivalent to an **lnj** or onj operation where the operands do not have common attributes. However, a **lnj** or onj operation between relations with common attributes can be as large as the cartesian product, though in many cases it may reduce the data volume.

The remote execution of an **alt** or **grp** operation could either decrease, increase, or not affect the data transfer volume. In most cases, **grp** would be expected to decrease the data transfer volume, and **alt** would be expected to increase it only marginally, if at all. The **trc** operation would also increase the data transfer volume (the amount depending on the size of the key attribute values in the operand relation since these values are repeated in the result) and the **tcr**

operation would decrease it.

Suppose that a set of non-permissible remote operation types is predetermined; for example, all lnj and onj operations equivalent to a cartesian product, trc operations, and alt operations with an add clause and no drop clause. Algorithm comp_site_set could simply force any operation node that has an operation matching any element of this set to have a NIL site set, and hence be performed as part of the final subquery at the query site. However, this strategy would not be optimal if a subsequent operation (corresponding to an adjacent or non-adjacent predecessor node) has a data volume reduction effect that more than offsets the data volume expansion effect of the non permissible remote operation.

An algorithm that attempts to consider subsequent operations when deciding whether the site set of particular operation node should be set to NIL would be much more complicated than the one presented here. Also, it must be remembered that data volume reduction/expansion effects are being predicted in a very coarse manner, without the use of relation profiles and estimates of actual result relation size. Therefore, the refinement of considering subsequent operations would probably not offer a consistent improvement.

A reasonable approach would be to include in the non-permissible remote operation set only those operation types

with a dramatic data volume expansion effect. These would clearly include the cartesian product. The other members would be other **onj** and **lnj** operations with a demonstrated high data expansion effect.

Determining these latter members would require keeping track of past results. For example, each time an **onj** or **lnj** operation specified in the database integration mapping is performed locally, the size of the result relation could be compared to the sum of the size of each operand, and if the result relation size is not larger, then the operation could be added to a permissible remote operation set (the operation could be identified by its operands and the join attributes). Of course, this assumes that operations are devised to calculate relation sizes.

Initially, all **onj** and **lnj** operations would be non-permissible (for remote execution) by default, but some would become permissible as the permissible remote operation set grows. Of course, changes in database contents could make past observations irrelevant, so global query delay times should be monitored for trends of increasing delays in order to determine whether all or some of the members should be removed from the permissible remote operation set. The full development of these ideas constitutes a further research topic (see Chapter 7).

5.5 DAG EQUIVALENCE TRANSFORMS

This Section discusses background considerations pertinent to the design of the improve DAG algorithm mentioned in Section 5.3. Two types of DAG equivalence transform which appear to offer practical ways of optimizing queries within the framework of our query processing algorithm are 1) generating early projection and selection operations and 2) eliminating duplicate subexpressions. These transforms will be discussed separately below, followed by a discussion of how they might be incorporated into an improve_DAG algorithm.

5.5.1 Generating Early Projection and Selection Operations

The idea of generating early projection and selection operations is related to the technique of optimizing relational algebraic queries in a centralized DBMS by deriving selection and projection operations on the operands of expensive binary operations from those specified on the results [ULLM 82]. This will be referred to as 'pushing' selections and projections (which may be collectively referred to as **lim** operations in our GQML syntax) past other operations. This type of transform is intended to result in a reduction of the size of binary operation operands and, therefore, the query execution delay. In a DDBMS this type of transform could also result in a reduction in the volume of data transfer if the operands are located at remote sites

[CERI 84].

Let us call the operation which we are attempting to push a **lim** operation past as the 'barrier operation'. Also, let us call the **lim** operation originally specified on the barrier operation's result relation as the 'original **lim** operation', and the new operations produced on the barrier operation's operand relations as the 'new **lim** operations'. A pushing operation is performed on a <barrier operation, original **lim** operation> pair. For the sake of the following discussions, we will allow a barrier operation to be either a binary or unary GQML operation which is not a **lim** operation. If the barrier operation is binary, a new **lim** operation could be defined on one or both of its operands by a pushing operation. Also, we will assume that the selection predicate of an original **lim** operation is in, or has been converted to, conjunctive normal form. A predicate in this form consists of the conjunction of predicates (called conjuncts) none of which contain conjunctions.

If a barrier operation's parameters (apart from the operand list) are changed by the pushing operation or if it is replaced by another non-**lim** operation by the pushing operation, then the barrier operation will have been 'modified'. The barrier operation will have been 'removed' if the result of the pushing operation does not include a non-**lim** operation. If the original **lim** operation is made redundant by the new **lim** operation(s) then the original lim

operation will have been 'removed'. On the other hand, a **lim** operation on the barrier operation result which is not identical to the original **lim** operation may be required after the pushing operation. In this case, the original **lim** operation will have been 'modified' by the pushing operation.

If the barrier operation is a **lnj** operation with a **where** and/or an **attrs** clause, then the original **lim** operation is represented by those clauses. A **lim** operation is not pushed past another immediately preceding (in execution order) **lim** operation. Instead, the two **lim** operations are merged into one whose predicate (**where** clause) is the conjunction of the two predicates, whose projection list (**attrs** clause) is that of the **lim** operation which would be the last executed (closest to the result node in a DAG representation), and whose operand is the operand of the **lim** operation which would be the first executed. The **lim** operation resulting from such a merge may then be original with respect to a non-**lim** barrier operation.

If after a pushing operation there are no new **lim** operations, and neither the original **lim** operation nor the barrier operation has been modified or removed, then that pushing operation will have 'failed'.

A pushing operation involving a **u**, **int**, or **dif** barrier operation will always remove the original **lim** operation and produce two new **lim** operations both of which are identical

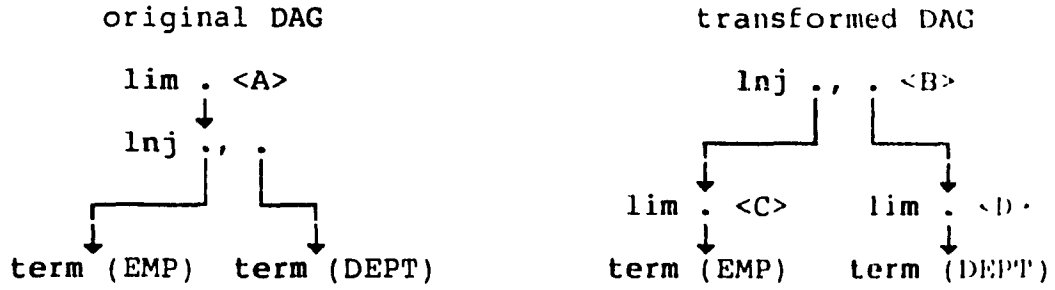
to the original. A pushing operation involving a **lnj** or **onj** barrier operation may remove the original **lim** operation, if the original **lim** projection list includes the join attributes, and each conjunct of the selection can be applied to an operand relation (a predicate can be applied to a relation if all attributes referred to in the predicate are in the relation's schema).

If neither of these conditions are met, a non-failure pushing operation on a **lnj** or **onj** barrier operation may still be possible. However, the original lim operation would be modified rather than removed, i.e. only some of the restrictions of the original **lim** operation may be passed to the new **lim** operations. Specifically, if the original **lim** operation projects out join attributes, then a modified original **lim** operation would be required to do so after the transformation since join attributes cannot be projected out by the new **lim** operations. Also, suppose that an original **lim** operation predicate conjunct cannot be applied to either join operand. Then that conjunct must be included in the modified original **lim** operation predicate.

The generation of new **lim** operations in terms of a DAG equivalence transformation would involve the addition of new operation nodes and the replacement or removal of others. Consider Figure 5.9. The original **lim** operation node is removed, the **lnj** operation node is modified and becomes the query result node, and two new **lim** operation nodes are added

which become subquery result nodes. It is required that the addition and removal of nodes from an operation DAG be accompanied by their addition and removal from the node list as well.

terminal relation schemata:
EMP(EMPID CITY EMPNM SAL DEPTID) Site 1
DEPT (DEPTID DEPTNM LOCATION) Site 2



<A> ::= **where** SAL < 100 **and** CITY <> LOCATION
attrs EMPID, EMPNM;

<B> ::= **where** CITY <> LOCATION **attrs** EMPID, EMPNM;

<C> ::= **where** SAL < 100 **attrs** EMPID, DEPTID;

<D> ::= **attrs** DEPTID;

Figure 5.9 example of a DAG equivalence transform

The only unary operators in the basic relational algebra are projection and selection. However, the GQML has other unary operators (**ren**, **alt**, **grp**, **trc**, and **tcr**). If any of these appear in a final subquery DAG as successors of a **lim** operation node it would be useful to involve them as barriers in pushing operations so that new **lim** operations are added to remote subquery DAGs, or at least so that sizes of the operands of expensive operations are reduced.

A **lim** operation on a **ren** operation result relation can be replaced by a **lim** operation on the operand relation. The new **lim** operation is identical to the original except that the inverse of the **ren** operation must be applied to the names of attributes in the projection list and the predicate. However, a **lim** operation is not as readily pushed through **alt**, **grp**, **trc**, or **tcr** operations. These cases are discussed separately below

5.5.1.1 Pushing With an alt Barrier Operation

A new **lim** operation on the operand of an **alt** operation cannot project out any attributes whose values are used in the **alt** operation to evaluate new (**added**) attributes. If such attributes are not dropped in the **alt** operation and are projected out by the original **lim** operation, the best that can be done is to add them to the **alt** operation drop list.

An original **lim** operation predicate conjunct can be moved unchanged to the new **lim** operation only if it does not incorporate attributes which are added in the **alt** operation. An original lim operation predicate conjunct which includes one or more added attributes may be moved as a transformed predicate by substituting the corresponding evaluation function for the attribute. For example, consider the

following operation:

```
lim
    alt EMP drop SAL1, SAL2 add SAL numeric 3 =
                                    max(SAL1, SAL2);
  where SAL = 100
;
```

An equivalent operation would be

```
alt
    lim EMP where max(SAL1, SAL2) = 100;
  drop SAL1, SAL2 add SAL numeric 3 = max(SAL1, SAL2);
;
```

Typically in integration mapping, an **alt** operation would be specified on the results of an **onj** operation on global relations (see Section 4.1). In these cases it is likely that a subquery would be defined for each of the onj operands (i.e. they would likely represent data from different sites). Therefore, it would be especially valuable to be able to derive new **lim** operations on the **onj** operands from any **lim** operation on the **alt** operation result relation. This would involve two pushing operations: one on the **alt** operation followed by one on the **onj** operation. However, it may be difficult to determine valid selection predicates for the **lim** operations on the onj **operands.**

For example, consider Figure 5.10 in which this type of transformation has been performed. Let us call the comparison operation in the original **lim** operation opl ,'=' in this example, and those in the new **lim** operations op2 and op3, both of which are '>=' in this example. The use of '=' for opl and op2 might appear to be a better choice since the new **lim** operations would be more restrictive, but this would

be incorrect. To see this, suppose that two tuples exist in 'EMP1' and 'EMP2' respectively with the same 'EMPID' value and the 'EMP1' tuple has a value of 100 for 'SAL1' while 'EMP2' has a value of 150 for 'SAL2'. Then the result relation tuple with that 'EMPID' value would have a 'SAL' value of 100 instead of 150 if '=' is used for op1 and op2.

Note that the original **lim** operation is required in the transformed expression. In [DAYA 85] new selections that do not replace the original selection are referred to as "partial" while those that do are referred to as "full."

```
terminal relation schemata:
     EMP1 ( EMPID  SAL1 )
     EMP2 ( EMPID  SAL2 )

original expression:

     lim
         alt
             onj EMP1, EMP2 ori ORIG;
          drop SAL1, SAL2
          add SAL numeric 3 =
            SAL1 if ORIG = EMP1 else
            SAL2 if ORIG = EMP2 else
            max(SAL1, SAL2);
       where SAL = 100
     ;


transformed expression:

     lim
         alt
             onj
                 lim EMP1 where SAL1 >= 100;,
                 lim EMP2 where SAL2 >= 100;
               ori ORIG;
          drop SAL1, SAL2
          add SAL numeric 3 =
            SAL1 if ORIG = EMP1 else
            SAL2 if ORIG = EMP2 else
            max(SAL1, SAL2);
       where SAL = 100
     ;
```

Figure 5.10 example of two pushing operations involving an onj barrier operation followed by an alt barrier operation

Although the issue here is how to perform a successful pushing operation where the **onj** operation is the barrier (rather than the **alt** operation) the difficulty arises because a selection is predicated on an attribute added in the **alt** operation. In other words, the difficulty is a side-effect of the previous pushing operation involving the **alt** operation.

It is possible to predetermine whether an op2 and op3 exists which allows this transformation given a particular function (**max** in the Example) and a particular op1, and whether the new selections (incorporated in the new **lim** operations) are full or partial [DAYA 83, DAYA 85]. For example, given an op1 of >, an op2 and op3 of > would permit the transformation. Furthermore, the original **lim** operation (with respect to the **alt** operation) can be removed. Thus the new selections are full. Note that if the functions **sum** or **avg** were used instead of **max** in the above example the second pushing operation (involving the onj operation as the barrier) would fail.

5.5.1.2 Pushing With a grp Barrier Operation

Three conditions limit the generation of a new **lim** operation on the operand of a **grp** operation, such that the new **lim** operation does not affect the result of the **grp** operation:

1) The new **lim** operation cannot project out attribute specified in the grp operation projection list,

2) The new **lim** operation cannot project out attributes referred to by added attribute evaluation functions in the **grp** operation,

3) If aggregate functions are used to evaluate added attributes in the **grp** operation, the exclusion of tuples

from the operand is not permitted. This means that the new **lim** operation cannot project out prime attributes, nor can it include a selection predicate that is not true for all tuples.

In a pushing operation involving a **grp** barrier operation a more restricting (in terms of projection and selection) new **lim** operation may be defined.

Suppose that the original **lim** operation projects out attributes added in the **grp** operation which are not referred to in its selection predicate. Then, the new **lim** operation may drop attributes used in evaluating these added attributes as long as this doesn't affect the evaluation of other added attributes. The **grp** operation would have to be modified to exclude the evaluation of the added attributes projected out by the original **lim** operation.

Suppose that the original **lim** operation projects out attributes included in the projection list of the **grp** operation, and does not refer to these in its selection predicate. Then, the new **lim** operation may project these attributes out too, as long as it doesn't affect the evaluation of added attributes which not projected out by the original **lim** operation or referred to in its selection predicate.

Conjuncts from the original **lim** operation predicate cannot be moved to the new **lim** operation if one or more

added attributes are evaluated with an aggregate function, and that added attribute is referred to in the original lim operation predicate or is not projected out by the original lim operation.

5.5.1.3 Pushing With a Transpose Operation Barrier

A selection predicate from an original **lim** operation on a **trc** or **tcr** result relation may only be added to a new **lim** operation on the **tcr** or **trc** operand if it incorporates only those attributes which must exist in both the operand and result relations (i.e. the 'a' attributes as described in Section 3.1.6). Also, these attributes may not be projected out by the new **lim** operation.

If the original **lim** operation on a **tcr** result relation projects out a row attribute created by the **tcr** operation, then the **tcr** operation may be modified to exclude that attribute in its **row** clause. In this case the new **lim** operation would have to include a selection predicate to exclude those tuples from which that row attribute gets its values. Conversely, if the original **lim** operation on a **trc** result relation rejects all tuples whose sequencing attribute values are included in a particular set of values then the corresponding attributes may be excluded from the **trc** operation **row** clause. Also, the new **lim** operation would have to project out those attributes.

To illustrate these points consider the following relations:

SALES1	EmpID	EmpNm	P1Sales	P2Sales	P3Sales
	100	"Smith"	53	49	51
	200	"Jones"	60	52	55

SALES2	EmpID	EmpNm	Period	Sales
	100	"Smith"	1	53
	100	"Smith"	2	49
	100	"Smith"	3	51
	200	"Jones"	1	60
	200	"Jones"	2	52
	200	"Jones"	3	55

As shown in Section 3.1.6 'SALES1' can be created by a tcr operation on 'SALES2', and 'SALES2' can be created by a trc operation on 'SALES1'. The following sequence of operations is equivalent to the **lim** operation on 'SALES1':

```
lim
    trc SALES2 col Sales
               to_row P1Sales, P2Sales, P3Sales
               seq Period;
  attrs P1Sales, P2Sales
;
```

An equivalent sequence of operations in which the lim operation is executed before the **trc** operation is as follows:

```
tcr
    lim SALES2 where Period = 1 or Period = 2;
  col Sales
  to_row P1Sales, P2Sales
  seq Period;
```

Now consider the following sequence of operations which is equivalent to the lim operation on 'SALES2':

```
lim
    trc SALES1 row P1Sales, P2Sales, P3Sales
      to_col Sales
      seq Period;
  where Period = 3
;
```

An equivalent sequence of operations in which the projection is executed before the **trc** operation is as follows (**alt** is used instead of **lim** for brevity):

```
trc
    alt SALES1 drop P3Sales;
  row P1Sales, P2Sales
  to_col Sales
  seq Period;
```

5.5.2 Eliminating Duplicate Subexpressions

In a query DAG, duplicate subexpressions may be exhibited as 1) duplicate subgraphs of separate subquery DAGs, 2) duplicate subquery DAGs, or 3) duplicate subgraphs of the query DAG. Eliminating duplicate subexpressions would consist of merging duplicate subgraphs. In the first case, eliminating duplicate subexpressions may not be useful since the subqueries themselves may end up being executed at different sites. In the second case, eliminating duplicate subexpressions would be very useful since this would eliminate unnecessary data processing and, if one or more remote subqueries are eliminated, inter site communications. In the third case, eliminating duplicate subexpressions would be definitely useful only if they encompass non-final subqueries and/or parts of the final subquery.

Finding all duplicate subgraphs in a query DAG is potentially very time consuming and most likely impractical in most cases. A practical algorithm might be restricted to finding duplicate subgraphs that encompass non-final

subqueries since these must have identical terminal nodes Thus, the search may be narrowed down considerably at the start.

5.5.3 DAG Improvement Algorithm Design Issues

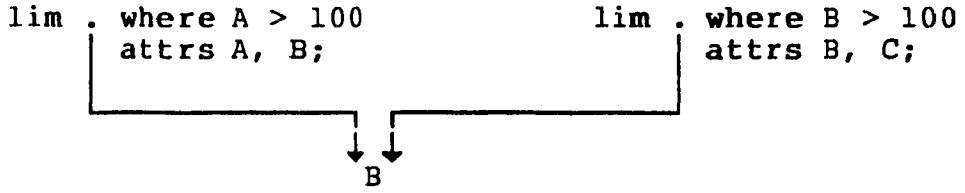
Suppose that a query DAG is a tree with no duplicate subexpressions. DAG improvement would include pushing **lim** operations as far as possible towards the terminal nodes, into the subqueries. The overall algorithm, shown in Figure 5.11, is simple, and the observations made in Section 5.5.1 show how the pushing operation could be applied to specific barrier operations. In this algorithm, the **where** and **attrs** clauses of a **lnj** operation are treated as though they were in a preceding **lim** operation node. As part of the pushing operation the current node becomes either the barrier operation node (possibly modified) or its replacement node.

```
algorithm push_to_terms
/* pushes lim operations to terminal nodes of
   query operation trees */

  input  : result node of operation tree
  output : result node of modified operation tree

  {
  if (current node is a lim operation) {
    while (successor node is a lim operation) {
      merge current and successor node and make this
      the new current node;

    if (successor node is non-terminal)
      perform pushing operation on current and
      successor node (barrier operation node),
      re-assigning the current node in the process;
    }

    for (each successor node of current node)
      call push_to_terms with successor node;
  }
```

Figure 5.11 algorithm to push **lim** operations towards terminal nodes of query operation trees

Suppose that the query DAG is not a tree, and that a barrier operation node exists which has multiple **lim** operation predecessor nodes, possibly generated by pushing **lim** operations towards, but not beyond, this operation node. It would be incorrect to perform a pushing operation independently involving the barrier node and a single predecessor node. For example, consider Figure 5.12 where 'B' represents the barrier operation node. A single original **lim** operation may be inferred from the multiple predecessor **lim** operation nodes whose projection list is the union of the multiple projection lists (an empty **attrs** clause is equivalent to a projection list containing all operand attributes), and whose predicate is the disjunction of the

multiple predicates (the predicate would then be converted to conjunctive normal form). The pushing operation could then be performed using this inferred original **lim** operation. Note that any of the multiple predecessor **lim** operation nodes may be modified or removed by the pushing operation. The inferred original **lim** operation can be no more restrictive than any of the **lim** operations it is derived from, and it is possible that it would be much less restrictive than any of them even to the point of being equivalent to a **lim** operation with no **attrs** or **where** clause.



inferred original **lim** operation:

```
lim <B> where A > 100 or B > 100 attrs A, B, C;
```

Figure 5.12 example of multiple predecessor **lim** operation nodes in pushing operation

If the query DAG has duplicate subgraphs, then a decision has to be made whether to merge the duplicate subgraphs before or after the pushing operations. If the pushing operations are performed first, then the subgraphs may be made different and, therefore, non-mergeable. If the pushing operations are performed last, then it may be that the generated new **lim** operations are so unrestrictive that it would have been better to use the separately transformed,

non-mergeable subexpressions. Without database statistics, it would not be possible to derive delay estimates upon which to base a choice, so the decision would have to be arbitrary or based on the observation of delays during actual use.

The above discussion shows that a thorough improve DAG algorithm based on **lim** operation pushing and duplicate subexpression elimination would be very complex, and, in the context of fast local communication links, probably impractical. Further research is required to determine whether a simplified and worthwhile algorithm based on these principles can be devised.

5.6 SELECTION OF REMOTE SUBQUERY SITES

In the case where the site set of a remote subquery result node contains more than one site id, the query processor must select one of those sites to further process and execute the subquery. In the remainder of this Section, multiple possible subquery sites for the same remote subquery will be referred to as 'candidate remote subquery sites' and the selected one as the 'remote subquery site'.

The goal of remote subquery site selection is to minimize the delay between the start of dispatching the remote subqueries and the end of receiving the subquery result data, so that the time during which the query site is waiting idle is eliminated or at least minimized. Thus query

and result data communication delay, the relative performance level of the DBMSs at the remote sites, and the nature of the remote subqueries themselves are pertinent to remote subquery site selection.

Suppose that the same data may be transferred between any two sites with the same delay (including communication session set-up delay). If all remote subqueries are sent as a group to a single site and the results are sent back as a group, then the communication session set-up delay will be minimized. If the remote subqueries are distributed among the maximum possible number of sites, then the set-up delay will be maximized. The total time required to actually transmit the data will be the same in both cases.

If we assume that all remote subqueries will involve the same processing and execution delay, regardless of the remote subquery sites, then the fully distributed approach will have the greatest degree of parallelism and the smallest maximum remote subquery processing and execution delay. If communication session set-up delay is insignificant compared to remote subquery processing and execution delay, then overall the fully distributed approach would be best. Otherwise, a less fully distributed or even a single remote site approach could be best if this means that sites with exceptionally slow DBMSs are avoided as remote subquery sites.

If we do not assume that communication delay between

each pair of sites is uniform and that all remote subqueries will involve the same processing and execution delay, then remote subquery site selection becomes very complex. Moreover, if local database statistics are unavailable, as we assume is the case, then it would not be possible to estimate the size of GQML operation operand relations and therefore, directly estimate remote subquery execution delay. Even if local database statistics were available, the query site would have to acquire knowledge of the remote operations required to materialize remote site local participant schema relations and the amount of local optimization which would be done at each candidate remote subquery site.

For the purposes of this thesis, an initial simple approach to remote subquery site selection which conforms to our initial assumptions is proposed. The development of more elaborate approaches is a future research topic.

Suppose that communication session set-up costs are insignificant, then the remote subquery site selection problem becomes one of distributing subqueries such that the maximum processing and execution delay among all subquery sites is minimized. Suppose further that given a particular subquery Q_i and a particular site S_j, a function yielding a relative delay value $RD(Q_i, S_j)$ may be computed at the query site. If Q_i is assigned to S_j, then $RD(Q_i, S_j)$ is greater than zero, otherwise it is zero. A simplified RD function

may, for example, use an approximate relative performance factor assigned to each <site, operation> pair and an relative cost factor assigned to each type of operation, which would also be very approximate since operand sizes would be assumed to be similar for each subquery. However, the relative operation cost factor would serve to discriminate between almost always relatively expensive operations such as cartesian product, tuple grouping, and transposition, and almost always relatively cheap operations such as projection.

The best distribution would be the one where

$$\underset{\text{all } s}{\text{MAX}} \left(\sum_{\text{all } q} RD(Q_q, S_s) \right)$$

is a minimum. All possible distributions can be computed using a variant of the algorithm for finding a minimum sized set of subquery sites described in [TEMP 83], in the context of the Mermaid system (in this version of the Mermaid system, communication costs are assumed to be dominant, and a minimum number of subquery sites is desired to minimize communication delay during execution of a semijoin reduction strategy). This algorithm will be explained through an example.

Suppose that candidate remote subquery sites have been computed as follows:

remote subquery	candidate sites
a	{S1}
b	{S2, S3}
c	{S3, S4}

Each candidate site set can be interpreted as a disjunction of remote subquery sites (e.g. subquery b can be sent to S2 or S3), and the conjunction of all the disjunctions represents all possible subquery distributions. For the example, we get the following expression:

$$(Sl_a) \text{ and } (S2_b \text{ or } S3_b) \text{ and } (S3_c \text{ or } S4_c)$$

The subscripts on the site id's indicate remote subquery site assignment. The above expression can be converted into the following disjunctive normal form expression:

$(Sl_a \text{ and } S2_b \text{ and } S3_c)$ or $(Sl_a \text{ and } S2_b \text{ and } S4_c)$ or

$(Sl_a \text{ and } S3_{bc})$ or $(Sl_a \text{ and } S3_b \text{ and } S4_c)$

The symbol '$S3_{bc}$' is derived from '$S3_b$ and $S3_c$'. Each conjunction of subscripted site numbers represents a possible distribution of subqueries. The algorithm for converting the initial expression into disjunctive normal form runs in exponential time, but this is not serious since the number of subqueries and the size of candidate site sets would normally be quite small for ad hoc queries. Finally, the maximum relative delay for each distribution would be computed and the distribution associated with the minimum value would be selected. In the event of a tie, one of the best distributions would be selected arbitrarily.

This site selection algorithm is summarized in Figure 5.13.

```
algorithm site_sel
/* select remote subquery sites */

  input:  set of remote subquery DAG result nodes with
          site sets evaluated

  output: set of remote subquery DAG result nodes, each
          one assigned to a remote subquery site

  {
  assemble initial conjunctive normal form distribution
  expression using site set contents of all nodes;

  convert distribution expression to disjunctive normal
  form;

  min rel_delay = high_value;
  for (each conjunctive term in distribution
expression)        {
    compute maximum relative delay (max_rel_delay);

    if (max rel_delay < min_rel_delay) {
      select current conjunctive term to represent the
      distribution;

      min rel_delay = max_rel_delay;
      }

    assign subqueries to sites according to final
    selected distribution;
    }
  }
```

Figure 5.13 remote subquery site selection algorithm

5.7 QUERY TRANSLATION

A subquery DAG must be translated back into GQML text if the subquery is remote, or into a representation that can be used by the local DBMS if the subquery is final or local.

5.7.1 DAG to GQML Text Translation

A remote subquery in GQML form consists of a result relation name and a single GQML operation or a single nested hierarchy of GQML operations, as does a user query. The result relation name can be obtained from the subquery DAG result node. The translation from a DAG to GQML text is easily implemented as a postorder traversal of the DAG where text is generated for each node, as shown in Figure 5.14.

```
algorithm DAG_to_GQML
/* translate DAG to GQML operation text */

input:   1) DAG result node or terminal node
         2) pointer to text file

output:  GQML text in text file

  {
  if (input node is terminal) {
    write relation name corresponding to node;
    return;
    }

  write operator name corresponding to node (u, dif,
  int, lim, etc.);

  for (each successor node of input node) {
    translate corresponding subgraph;
    (algorithm DAG_to_GQML)

    if (all successors not yet considered) write ',';
    }

  write remaining operation clauses associated with
  input node, if any (e.g. where clause, attrs clause);

  write ';';
}
```

Figure 5.14 DAG to GQML text translation algorithm

5.7.2 DAG to Local Representation Translation

The HDDBMS is just another user to the local DBMS, so the interface(s) made available to any user by the local DBMS influences the method for translating a final or local subquery DAG into an equivalent local representation.

For example, suppose that the local DBMS interprets command programs written to a text file, and that it is possible to generate a command program segment equivalent to any GQML operation. Then the translation may be implemented as a postorder traversal of the DAG, with a program segment generated for each node that refers to entities in the local host schema. If the local DBMS is not relational or tabular, then a mapping between the translated local host and the local host schemata would be required in program generation (such mapping and program generation is a future research topic, as mentioned in Chapter 7). All program segments would be written to the same file to create a single command program which could then be submitted to the local DBMS interpreter. Algorithm DAG_to_local_com_prog (Figure 5.15) illustrates this approach.

Rather than submit each local subquery command program separately to the local DBMS interpreter, it might be possible to concatenate them in the order in which they were generated to create a single command program representing all local subqueries. This command program would be submitted once to the interpreter. This could reduce delay

if the interpreter does not reside in main memory between uses.

Algorithm DAG_to_local_com_prog is made specific to a particular (command program driven) local DBMS by the DBMS-specific code generation module included in the inputs. This module is used to generate the actual program segment from the current operation node contents, and it would refer to succeeding nodes in the DAG for operand relation names. An operand relation would be a remote subquery result relation, a local subquery result relation, a translated local host schema relation, or an intermediate relation created by a local subquery (this would occur when the subquery being translated is also local and has a common subgraph with another local subquery).

After the code corresponding to an operation node is generated, the node is marked as 'visited', as discussed in Section 5.3. To recapitulate, if such a node is encountered when traversing the DAG, it is assumed that the corresponding relation has been materialized at the local site and is in local DBMS format. Let us define duplicate code as code which results in the materialization of the same relation. Then marking nodes as visited prevents the generation of code which is a duplicate of that existing earlier in the command program, or in the command programs for other local subqueries.

```
algorithm DAG_to_local_com_prog
/* translate DAG to local DBMS command program */

input:    1) DAG result node or terminal node
          2) DBMS-specific code generation module
          3) local site id
          4) query site id
          5) pointer to text file

output:  local DBMS command code in text file

{
if ( input node
       1) not terminal and
       2) not a subquery result node and
       3) not marked as visited )
  {
  for ( each successor node of input node )
     translate corresponding subgraph;
     (algorithm DAG_to_local_com_prg)

  write local language command program segment
  corresponding to input node operation using
  the DBMS-specific code generation module

  (if input node is result node of a final subquery
   and local site id not the same as query site id
   include reformatting of results to global format)
  ;

  mark input node as visited;
  }

if (input node is the result node of a remote subquery
    and node not marked as visited)
  {
  write commands to reformat the received global format
  data file to local format using the DBMS-specific
  code generation module;

  mark input node as visited;
  }

if (input node is the result node of a final subquery
    and local site id same as query site id)
  write ancillary commands for results presentation,
  etc. using the DBMS-specific code generation module;
}
```

Figure 5.15 DAG to local command program translation algorithm

In the design of DAG_to_local_com_prg, it is assumed that the local DBMS has the necessary data reformatting capabilities. The conversion of the current subquery results from local to global format is incorporated in the code segment which represents the last operation of a final subquery for a query submitted from a remote site (for a given local DBMS this may save a pass through the result data). Also, code is generated to convert remote subquery results from global to local format if this has not already been done.

Finally, if the subquery being translated is the final subquery of a locally submitted query, ancillary code for data presentation, presenting options to erase or move the result file, etc. are written. This would be done even for a null final subquery.

In cases where the local DBMS does not provide a command program interpreter, the DBMS-specific code generation module might have to generate code in a general purpose language with embedded local DML commands, and then have this program compiled and linked. A library of procedures and functions may be developed to simplify this task. This program would then be executed at the appropriate time. Some GQML operations might correspond to a single local DML command and these could still be handled by a local DBMS single-command interpreter.

Algorithm DAG_to_local_com_prg would produce a result

relation corresponding to each DAG node. Certainly the subquery result relation is required, as is any intermediate result relation which is positioned as an operand for another subquery (in the case of subqueries with overlapping DAGs). However, if the materialization of other intermediate result relations can be avoided by combining the operations represented by multiple nodes into a single operation, considerable time may be saved. In particular, a **lim** operation occurring before or after another operation can be combined with that operation in some DMLs since the additional processing simply consists of checking a record on input or output against a projection list and selection predicate.

This type of local optimization may be implemented by a local DBMS-specific pre-translation scan of a subquery DAG which would merge adjacent nodes into a single node where possible. The scan would have to locate nodes whose result relation would be an operand of another subquery and avoid merging such nodes with their predecessor nodes in the subquery DAG. Such nodes, however, may be merged with their successor nodes.

CHAPTER 6 HDDBMS PROTOTYPE IMPLEMENTATION

An HDDBMS prototype, called the Multiple Database Access System (MDAS), has been implemented in order to demonstrate the approach to database integration and query processing discussed in previous Chapters. In the remainder of this thesis, 'MDAS' will refer specifically to the current version of the system. Further development of the system beyond the limitations mentioned in this Chapter is discussed in Chapter 7.

The MDAS integrates two sites, represented by two PCs connected by an RS-232C null modem cable. One site has dBase III as its host DBMS, and the other site uses KnowledgeMan (KMan). All data processing, including data file reformatting, is performed by these DBMSs under the direction of the MDAS.

The MDAS accommodates the schema architecture described in Section 2.1. The global model is relational and a subset of the GQML described in Chapter 3 has been implemented as part of the system.

The MDAS provides each site with a relational integrated schema of the local host databases. The support of non-relational external schemata has not been implemented, so the integrated schema also functions as the external schema. Users can submit ad hoc global queries consisting of a result relation name and a single GQML

operation or hierarchy of GQML operations. The result of a global query is a file in the local host DBMS database file format.

Both KMan and dBase III store data in tables and, as such, may be considered relational from a structural point of view. Therefore, at both sites the translated local host schema and the local host schema are the same. The support for non-relational or non-tabular local host schemata has not been implemented.

6.1 PROTOTYPE ARCHITECTURE

The overall prototype architecture is illustrated in Figure 6.1. Overall, the MDAS may be considered as being composed of a remote request server, a terminal monitor, a global query processor, and a command program generator at each site. A file transfer module (not included in Figure 6.1) based on a multiple-file transfer version of the XMODEM protocol [CAMP 87] was implemented in order to support inter-site communication. All MDAS modules and the file transfer module were written in the 'C' programming language.

The user executes a terminal monitor process which requests the result relation name and the GQML query text. The GQML implementation is described in Section 6.2. The query text may either be entered directly at the terminal or submitted from a text file. The user must also ensure that

the remote request server process is executing on the other machine when a query is submitted. The global query processor transmits all remote subqueries in a single communication session with the remote request server process at the other site (the remote request server process busy waits for remote subqueries). The multiple file transfer protocol allows each remote subquery to be transferred as a GQML query in its own ASCII text file. This simplifies the implementation of the remote request server since it does not have to extract separate subqueries from the same file.

The terminal monitor writes the user query to a file in a designated directory and passes the global query processor the name of the query file and the local site id (a character string). Similarly, when a remote request server has received the text of a remote data request from the opposite site , it executes a local instance of the global query processor and passes it the name of the file containing the text and the opposite site's id. A global query processor accesses initialized data which includes its own site id, so by comparing the received site designator with its own, it can determine whether or not the query has originated at the same site.

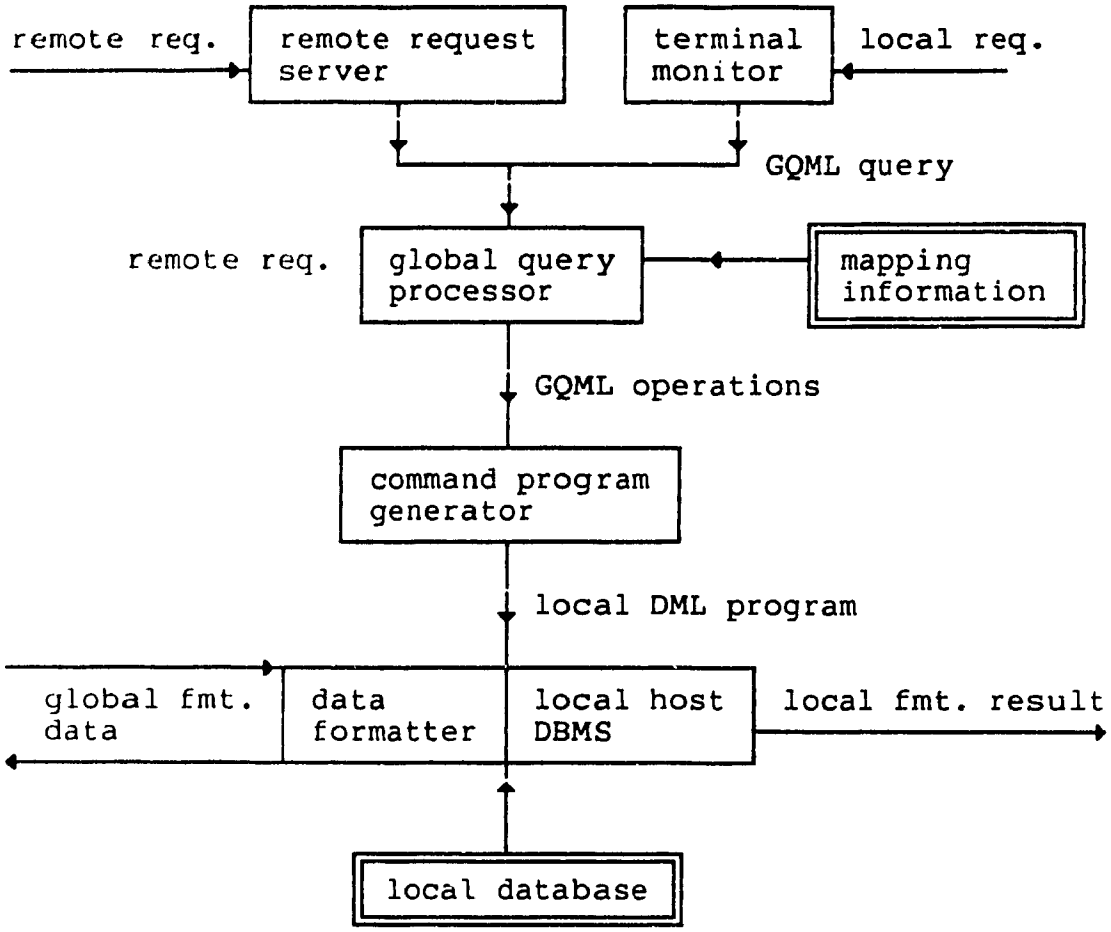


Figure 6.1 MDAS architecture

The result to a user's query is ultimately assembled in a file in the local host DBMS database file format having the result relation name and residing in a predetermined directory. The global query processor instructs the local host DBMS to offer to display the results at the terminal when the data processing is complete. In contrast, the result to remote request is ultimately assembled in a global format ASCII text file (by the local host DBMS under the instruction of the global query processor) and is then sent

back to the query site by the remote request server. The results of all remote subqueries which were received as a set are set back in the same multiple file transfer session.

Since all communication between sites consists of ASCII text file transfer, the MDAS may be adapted to a commercial local area network connection by replacing the file transfer module by an appropriate network interface.

At both sites, the local host DBMS performs data processing by running command programs written by the command program generator. The local DBMS kernel is loaded and run as a child process (with the command program file as a parameter) by the global query processor. For each query, the DBMS kernel needs to be loaded at least once to execute the final subquery. If a given query also has local subqueries, the kernel needs to be loaded once to execute these before receiving the remote subquery results and executing the final subquery (all local subquery command programs are written to the same file).

The file transfer module is also loaded into memory and run as a child process when required by either the remote request server or the global query processor. Admittedly, loading both the DBMS kernel and the file transfer module twice at the query site for one query (required if there are remote subqueries) adds significantly to the response time. Keeping the DBMS kernel and the file transfer module in main memory between uses would have been preferable, but not

possible with current resources. First, neither dBase III nor KMan (or at least the versions of these systems in our possession) provides the required interface, nor did we have access to the source code to create the required interface. Secondly, the operating system used (PC-DOS) manages a maximum of 640 K of main memory, which is also the size of main memory on available machines. This is too limited to simultaneously accommodate the MDAS, the file transfer module, and either local DBMS.

6.2 GQML IMPLEMENTATION

6.2.1 Implementation Scope

There are four domain types in the MDAS designated by the keywords **char**, **numeric**, **logical**, and **date**. Domain specifications involving the first three domain types are as described in Section 3.2. The **date** keyword directly specifies a domain without any added parameters, as is the case with the **logical** keyword.

Arithmetic and logical operators, and comparators have been included in the GQML implementation, as described in Section 3.2. Operations other than comparisons on **date** and **char** values are not included. No built-in functions are provided so, for example, operations such as domain conversion, value decomposition, and aggregate statistics computation are not supported. All of the relational

operations except **ren** and **grp** are implemented. Attribute renaming can be handled with **alt**, and without aggregate functions, there was little point in supporting **grp**. Although many string, date, domain conversion, and aggregate operations are supported by both KMan and dBase III, and so could have been included in the GQML, they were excluded because of time limitations.

The domain types available in the MDAS clearly reflect the field types of the local host DBMSs, KMan and dBase III. Both have direct counterparts of the **char**, **numeric**, and **logical** types, dBase III has a direct counterpart of the **date** type, and KMan has functions to convert between a numeric Julian date and a character string representation having a format similar to the **date** type.

Both dBase III and KMan use default values rather than distinct null values for attributes which have not been formally assigned a value. Specifically, both use zero for **numeric** fields, a string of length zero for **char** fields and FALSE for **logical** fields. Thus the **ori** clause of the **onj** operator is an essential feature in the MDAS. In dBase III the **date** default value consists of blank day, month and year subfields and as such does not correspond to any 'real' date value, but this is the only case where the default value can function as a distinct null value.

6.2.2 Base Relation Declaration

Since the MDAS currently handles only two sites, base relation location information is incorporated simply by declaring base relations as either 'local' or 'remote'. A local base relation is one where the corresponding data is contained in the local host database, while a remote base relation is one where the corresponding data is only available from the remote host database. Consequently, the implemented syntax for base relation declarations uses the keywords **local** and **remote** where **base** is used in the description in Section 3.4.

In a system that handles more than two sites, location information would be independent of base relation declaration, and would be obtained either from a directory or through intersite dialogue, as discussed in Section 5.4. The incorporation of more than two sites is a topic for further work and would be accompanied by the implementation of a site selection algorithm.

In the MDAS, each base relation declaration must reside in its own text file in a designated subdirectory. The name of the text file must match the name of the declared relation. For example, the following local declaration would reside in a file named "TAB":

```
local TAB
    key CustName char 15,
    Branch char 10,
    Balance numeric 6;
```

A remote base relation declaration has the same syntax, except that the keyword **remote** is used instead of **local**. As is the case with GQML text, the system ignores unnecessary spaces, tabs and newlines when parsing a base relation declaration. Relation and attribute names may be a maximum of 8 alphanumeric characters long, and the first character must be alphabetic.

The MDAS allows the specification of one or more primary key attributes in a base relation declaration with the **key** keyword. In the above example, the 'CustName' attribute is specified as the primary key of the TAB relation. The MDAS infers the primary key attribute(s) of the result of GQML operations from the primary key attributes of the operand(s). In an **alt** operation the user may specify a new attribute with key status. With knowledge of key attributes the code generation functions can often generate more efficient local processing instructions than without.

6.2.3 Virtual Relation Definitions

A virtual relation definition consists of a text file containing GQML operation text for one GQML operation or one hierarchy of nested GQML operations. The name of the text file is the name of the virtual relation, so the virtual relation name and the '==' symbol is not included in the GQML text.

For operands, GQML operations can use relation names corresponding to text files containing either the text of other GQML operations or base relation declarations. In the following example, the virtual relation defined by the text in file R4 is the same as that defined by the text in file R3.

```
file R4:
     div
       lim TAB attrs CustName, Branch;,
       lim BRANCHES where Location = "NORTH" attrs Branch;
       ;

file R3:
     div R2, R1;

file R2:
     lim TAB attrs CustName, Branch;

file R1:
     lim BRANCHES where Location = "NORTH" attrs Branch;
```

Base relation declaration and virtual relation definition text files for a given site reside in the same designated directory. The total collection of these files correspond to the 'mapping information' module in Figure 6.1.

6.2.4 Database Integration in the MDAS

Database integration is accomplished through the creation of an appropriate set of base relation declaration and virtual relation definition text files. Consider Figure 6.2 which corresponds to a subset of Figure 3.1 to 3.5 and involves only two translated local host schema relations

('DEPT' and 'DEPTS').

At each site, there is a local base relation declaration file representing the local participant schema, and a virtual relation definition file representing the mapping from the translated local host schema to the local participant schema. At Site 1, the global schema is represented by the 'DEPT1' virtual relation definition file and the 'DEPT2' non-local base relation declaration file. At Site 2, the global schema is represented by the 'DEPT2' virtual relation definition file and the 'DEPT1' non-local base relation declaration file.

In a more elaborate system, each site would have direct access to a base relation declaration, in some form, for each global relation. This would provide a means of checking the validity of changes to mappings from translated local host schemas to local participant schemas, and provide an independent basis for higher level mappings.

```
translated local host schemata:

  file DEPT (Site 1):                file DEPTS (Site 2):

    local DEPT                         local DEPTS
      key DID char 5,                    key DNO char 5,
      NAME char 15,                      DNM char 15,
      BLDG char 15;                      LOC char 15;


mapping from translated local host schemata to local
participant schemata:

  file DEPT1 (Site 1):

    alt DEPT
      drop DID, NAME, BLDG
      add
        key DEPTID char 5 = DID,
        DEPTNM char 15 = NAME,
        STREET char 15 = BLDG,
        CITY char 15 = "MONTREAL";

  file DEPT2 (Site 2):

    alt DEPTS
      drop DNO, DNM, LOC
      add
        key DEPTID char 5 = DNO,
        DEPTNM char 15 = DNM,
        STREET char 15 = LOC,
        CITY char 15 = "TORONTO";


global schemata base relation declarations:

  file DEPT2 (Site 1):               file DEPT1 (Site 2):

    remote DEPT2                       remote DEPT1
      key DEPTID char 5,                 key DEPTID char 5,
      DEPTNM char 15,                    DEPTNM char 15,
      STREET char 15,                    STREET char 15,
      CITY char 15;                      CITY char 15;


mapping from global to integrated schema:

  file DEPTMNT (Site 1 and Site 2)

    u DEPT1, DEPT2;
```

Figure 6.2 MDAS base relation declaration and virtual relation definition mapping files for a simple integration example

The global schema to integrated schema mapping as well as the integrated schema itself at each site are represented by a copy of the same 'DEPTMNT' virtual relation definition file. Since the integrated schema is also the external schema in the MDAS, declarations for integrated schema relations are not required as an independent basis for higher level mappings.

6.3 MDAS QUERY PROCESSING

The global query processor and the command program generator together represent a limited implementation of the query processing algorithm described in Chapter 5.

As an initial effort, this implementation accommodates only two sites and as such does not include a remote subquery site selection algorithm (remote subqueries can only be dispatched to the one remote site). In query DAG generation, the terminal node declaration types (local or remote) are used to determine subquery DAG result nodes. An operation node is declared local (remote) if all of its successor nodes are all local (remote), otherwise it is declared non-single-site. Also, the current system does not differentiate permissible and non-permissible remote operations (see Section 5.4), nor does it perform DAG improvement equivalence transformations (see Section 5.5.). The implementation of these capabilities are topics for further work. However, since the program follows the

framework described in Chapter 5, these capabilities will be simply added as further 'C' functions, with very little change to the existing software.

The DBMSs perform all data file reformatting. In the global data file format all data is expressed as ASCII characters, attribute values are delimited with commas, tuples are delimited with a carriage return and a line feed, **char** and **date** values are enclosed by double quotes, and **logical** values are expressed as TRUE or FALSE. The order of the attributes in received data must match the order of the attributes in the corresponding base relation declaration. Thus knowledge of the operations that materialize a global relation and their effect on attribute ordering is currently required in specifying the corresponding base relation declaration. The accommodation of a standard data interchange format in which the order of attributes is indicated as part of the transmitted data would require the implementation of an independent formatting module.

The command program generator module follows algorithm DAG_to_com_prog (Figure 5.15) closely. The same top-level program for traversing the input subquery DAG is used at both sites. However, this program calls a site-specific code generation function which writes a segment of code corresponding to the operation node passed to it. Site-specific functions are also provided for generating code for file reformatting and the display of query results. Local

optimization in the form of node-merging to avoid producing unnecessary intermediate result relations (as described in Section 5.7) has not been implemented. Although very similar in principle and overall capability, the dBase III and KMan command programming languages differ considerably in syntax, and the basic DML commands do not correspond in a one-to one fashion. Therefore, the site-specific code generation functions are quite different.

CHAPTER 7 CONCLUSION AND FURTHER WORK

This thesis has shown that a global query and mapping language based on the relational algebra (the GQML) is a powerful tool for integrating pre-existing databases. Furthermore, algorithms have been designed and demonstration versions have been implemented to process GQML queries and provide some HDDBMS capabilities in the MDAS prototype front-end system.

These algorithms exploit the fact that a GQML query on virtual relations defined by GQML operations, and on declared terminal relations, is readily converted into an operation DAG. This operation DAG may then be decomposed into subgraphs representing remote, local, and final subqueries, on the basis of the site location of data for the terminal relations. The remote and local subqueries may be executed in parallel to produce intermediate results, and the final subquery on these intermediate results may then be executed to produce the final result relation.

Since the MDAS processes a remotely submitted query on a local participant schema with much of the same software that it uses to process a locally submitted query on an integrated schema, a local participant schema is readily supported as a view of the local host schema, defined by GQML operations on the local host schema relations. This mapping level is not supported in other HDDBMS designs

encountered in the literature, but it is included in our design primarily because we feel that it may be useful in absorbing some changes to local host schemata, thereby insulating the global schema and higher mapping levels from those changes.

The current design emphasizes practicality over optimization. In particular, the global query processing algorithm does not attempt to optimize the degree of parallelism in local query processing and execution. In fact, a query that can be executed at a single site (entirely as a 'final subquery') will be executed this way, with no parallel processing at all. Parallelism optimization would require estimates and comparisons of the delay involved in individual operations and global knowledge of the additional mapping operations required to materialize remote global schema relations. This in turn would increase the complexity and size of the system considerably. Furthermore, simple local DBMSs of the type that we assumed in our design and incorporated in our implementation do not maintain the database statistics required for operation execution delay estimation.

However, a number of ideas for delay reduction measures which do not require database statistics or global knowledge of local low-level mappings have been discussed in this thesis, namely the determination of permissible versus non-permissible remote operations (Section 5.6), generating

early projection and selection operations (Sections 5.5.1, 5.5.3), and eliminating duplicate subexpressions (Sections 5.5.2, 5.5.3). These ideas could be developed and implemented as natural extensions to the existing MDAS, and their usefulness in terms of reduced query execution delay as compared to increased program size and increased query processing delay could then be assessed.

The current design and MDAS implementation can function as a test bed for further work in other areas, including the following:

- The incorporation of non-relational/non-tabular local DBMSs, particularly those conforming to the hierarchical and network data models. This would go beyond existing work in the area of supporting relational views over such DBMSs in that the GQML includes operations which are not part of the basic relational algebra.

- The support of non-relational external schemata and query languages, as well as relational external query languages other than the GQML. One immediate possibility is the support of a natural language interface developed in a previous project that refers to an entity-relationship type database description and produces queries in a predicate calculus based language [DESA 88].

- The efficient processing of multiple subqueries submitted as a group from the same remote site. As mentioned in

Section 5.3, such subqueries may have common subexpressions which should preferably be executed only once. The subqueries could be generated such that the common subexpressions are separate subqueries, and a front-end at the remote site would then have to submit the subqueries in the correct order, and request that results for common subexpressions be retained so that they may be used as operands for the remaining subqueries. Alternatively, the query processing algorithm could generate the DAGs for all subqueries from the group before executing any of them. This would allow the algorithm to consider already generated DAGs when generating the DAGs for the remaining subqueries so that intersection DAGs are created only once. The algorithm would then have to execute multiple final subqueries.

- The development of local optimization techniques, possibly including the approach based on merging adjacent DAG nodes into a single operation as discussed in Section 5.7.2.

- The support of more than two sites. This would entail the development of a site selection algorithm, possibly the one outlined in Section 5.6, as well as the development of an approach to acquiring data location information as discussed in Section 5.4. Also, the existing file transfer module would have to be replaced by a local area network interface.

- The development of a structured data interchange format (SDICF) or incorporation of an existing SDCIF. This would probably necessitate the implementation of data reformatting

modules.

The support of global updates. This task would consist of two parts:

1) Determining when and how updates may be mapped across schema levels (this problem was introduced in Section 2.4.1). Perhaps the extension of the relational global data model with further constructs will be useful. This problem is complicated by the fact that an update which is mapped to a remote global schema relation still has to be mapped to the local host schema at the remote site.

2) Developing methods of global concurrency and recovery control, perhaps following the research presented in [GLIG 84] and [GLIG 86] as summarized in Sections 2.4.2.1 and 2.4.2.2. Note that for these ideas to be implemented, local DBMSs with transaction managers need to be incorporated into the system (the current local DBMSs are single-user with no transaction managers), and some source code access may be necessary for global recovery control implementation.

An update on a replicated global schema relation would have to be propagated to all copies of that relation. The potential conflict between local updates and maintaining the validity of integration mappings is a related problem for which the only practical solution appears to be the assertion of additional integrity constraints on local updates, at the expense of reduced local autonomy.

- The development of computer aids for translating local host schemata into the global data model and generating database integration operations. Such aids would be necessary for integrating realistically complex databases, as opposed to integrating small, example databases representing only a few entities. Some work has been done in this area, as exemplified by the Schema Integration System (SIS) project. A prototype for this system is described in [SOUZ 86] as providing interactive tools for deriving schemata in a common data model from pre-existing database schemata, and then quantifying the similarity between the structures and names used in the common model schemata in order to find semantic overlap. The common data model is quite similar to the relational model with some added constructs, including one for capturing generalization hierarchies. Much more work is required to develop a system that will progress beyond finding schema similarities and differences to pinpointing specific conflict types and generating mapping operations.

REFERENCES

[BATI 84] C. Batini and M. Lenzerini, "A Methodology for Data Schema Integration in the Entity Relationship Model," IEEE Transactions on Software Engineering, Vol. SE-100, No. 6, (Nov. 1984), pp. 651-663.

[BATI 86] C. Batini, M. Lenzerini, and S.B. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration," ACM Computing Surveys, Vol. 18, No. 4, (December 1986), pp.324-364.

[BREI 86] Y. Breitbart, P.L. Olson, and G.R. Thompson, "Database Integration In a Distributed Heterogenous Database System," Proc. IEEE International Conference on Data Engineering, (1986), pp.301-310.

[CAMP 87] J. Campbell, "C Programmer's Guide To Serial Communications," Howard W. Sams & Company, Indianapolis, (1987).

[CARD 87] A.F. Cardenas, "Heterogeneous Distributed Database Management: the HD-DBMS," Proceedings of the IEEE, Vol. 75, No. 5, (May 1987), pp.588-600.

[CHIA 81] W.P. Chiang, D. DeSmith, D. Leder, D.K. Nguyen, R. Perreault, and J. Fry, "Draft Specifications for a Structured Data Interchange Form," University of Michigan Database System Research Group, Working Paper 80 DI 2.6 (February 1981).

[CODD 72] E.F. Codd, "Relational Completeness of Data Base Sublanguages," in E. Rustin (ed.), "Data Base Systems," Prentice-Hall, Englewood Cliffs, New Jersey, (1972), pp.65-98.

[CODD 79] E.F. Codd, "Extending the Database Realational Model to Capture More Meaning," ACM Transactions on Database Systems, Vol. 4, No. 4, (December 1979), pp.397-434.

[CERI 84] S. Ceri and G. Pelagatti, "Distributed Databases - Principles and Systems," McGraw-Hill Inc., New York, (1984).

[DANI 82] D. Daniels, P.G. Selinger, L.M. Haas, B.G. Lindsay, C. Mohan, A. Walker, and P. Wilms, "An Introduction to Distributed Query Compilation in R*", Proc. Second International Conference on Distributed Databases, Berlin, (Sept. 1982).

[DATE 86] C.J. Date, "An Introduction to Database Systems," Volume 1, fourth edition, Addison-Wesley Publishing Company, Reading, MA, (1986).

[DAYA 78] U. Dayal and P. Bernstein, "On the Updatability of Relational Views," Proc. Fourth International Conference on Very Large Databases (1978), pp. 368-377.

[DAYA 83] U. Dayal, "Processing Queries Over Generalization Hierarchies in a Multidatabase System," Proc. Ninth International Conference on Very Large Databases (1983), pp. 342-353.

[DAYA 84] U. Dayal and H. Hwang, "View Definition and Generalization for Database Integration in a Multidatabase System," IEEE Transactions on Software Engineering, Vol. SE-10, No. 6, (November 1984), pp.628-645.

[DAYA 85] U. Dayal, "Query Processing in a Multidatabase System," in W. Kim, D.S. Reiner, and D.S. Batory (eds.), "Query Processing in Database Systems," Springer-Verlag, New York, (1985), pp. 81-108.

[DEEN 85a] S.M. Deen, R.R.Amin, and M.C. Taylor, "Query Decomposition in PRECI*," in F.A. Schreiber and W. Litwin (eds.), "Distributed Data Sharing Systems," Elsevier Science Publishers, (1985).

[DEEN 85b] S.M. Deen, R.R. Amin, G.O. Ofori-Dwumfuo, and M.C. Taylor, "The Architecture of a Generalised Distributed Database System - PRECI*," The Computer Journal, Vol. 28, No. 3, (1985), pp.282 290.

[DEEN 87] S.M. Deen, R.R. Amin, and M.C. Taylor, "Data Integration in Distributed Databases," IEEE Transactions on Software Engineering, Vol. SE 13, No. 7, (July 1987), pp.860-864.

[DESA 88] B.C. Desai, R.J. Pollock, and P.J. Vincent, "A Natural Language Interface To a Multiple Databased Office Information System," SIGOIS Bulletin, Vol. 9, No. 4, (Oct 1988), pp. 19-33.

[ELMA 81] R. Elmasri, C. Devor, and S. Rahimi, "Notes on DDTS: An Apparatus for Experimental Research in Distributed Database Management Systems," ACM SIGMOD Record, (July, 1981).

[EPST 80] R. Epstein and M. Stronebraker, "Analysis of Distributed Database Processing Strategies," Proc. Sixth International Conference on Very Large Databases, (1980), pp. 92-101.

[FERR 82] A. Ferrier and C. Stangret, "Heterogeneity in the Distributed Database Management System SIRIUS-DELTA," Proc. Eighth International Conference on Very Large Databases, (1982), pp. 45-53.

[GLIG 84] V.D. Gligor and G.L. Luckenbaugh, "Interconnecting Heterogeneous Database Management Systems," IEEE Computer (January 1984), pp. 33-39.

[GLIG 86] V.D. Gligor, "Transaction Management in Distributed Heterogeneous Database Management Systems," Information Systems, Vol.11, No. 4, (1986), pp. 287-297.

[GOOD 81] N. Goodman, P.A. Bernstein, E. Wong, C.L. Reeve, and J.B. Rothnie, "Query Processing in SDD-1: A System for Distributed Databases," ACM-TODS, Vol. 6, No. 4, (1981).

[GRAY 84] P.M.D. Gray, "Logic, Algebra and Databases," Ellis Horwood Limited, West Sussex, England, (1984).

[KAY 75] M.H. Kay, "An Assessment of the CODASYL DDL for Use With Relational Subschema," in Douque and Nijssen (eds.), "Data Base Description," North-Holland, (1975), pp. 199-214.

[KENT 82] W. Kent, "Choices in Practical Data Design," Proc. Eighth International Conference on Very Large Data Bases, Mexico City, (1982), pp.165-180.

[KORT 86] H.F. Korth and A. Silberschatz, "Database System Concepts," McGraw-Hill Book Company, New York,(1986).

[LAND 82] T. Landers and R.L. Rosenberg, "An Overview of Multibase," in H.J. Schneider, ed., "Distributed Databases", North Holland Publishing Company, (1982), pp.556-579.

[LINN 88] V. Linnemann, K. Kuspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Sudkamp, G. Walch, and M. Wallrath, "Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions," Proc. Fourteenth International Conference on Very Large Databases, Los Angeles, (1988), pp.294-305.

[MCLE 80] D. McLeod and D. Heimbeigner, "A federated architecture of data base systems," AFIPS Conference Proceedings, Vol. 39.

[NAVA 84] S.B. Navathe, T. Sashidar, R. Elmasri, "Relationship Merging in Schema Integration," Proc. Tenth International Conference on Very Large Data Bases, (1984), pp. 79-91.

[PIRO 82] A. Pirotte, "A Precise Definition of Basic Relational Notions and of the Relational Algebra," SIGMOD Record, Vol. 13, No. 1, (September, 1982), pp.30-45.

[ROSE 82] A. Rosenthal, D.S. Reiner, "Querying Relational Views of Networks," in W. Kim, D.S. Reiner, and D.S. Batory (eds.), "Query Processing in Database Systems," Springer-Verlag, New York, (1985), pp. 109-124.

[SCHK 81] M. Schkolnick and P. Sorenson, "The Effects of Denormalization on Database Performance," IBM Research Report RJ3082, (March 1981).

[SENK 73] M.E. Senko, E.B. Altman, M.M. Astrahan, and P.L. Fehder, "Data Structures and Accessing in Data Base Systems," IBM Systems Journal, Vol. 12, No. 1 (1973).

[SMIT 87] J.M. Smith and D.C.P. Smith, "Database abstractions: Aggregation and Generalization," ACM Transactions on Database Systems, Vol. 2, No. 2, (June 1977), pp.105-133.

[SMIT 81] J.M. Smith, P.A. Bernstein, U.Dayal, N. Goodman, T. Landers, K.W.T. Lin, and E. Wong, "Multibase integrating heterogeneous distributed database systems," AFIPS Conference Proceedings Vol. 50 (May, 1981), pp.487-499.

[SOUZ 86] J.M. Souza, "SIS - A Schema Integration System," in E.A Oxbrow (ed.) Proc. Fifth British National Conference on Databases, Canterbury, (July 1986), pp. 167-185.

[TEMP 83] M. Templeton, D. Brill, A. Hwang, I. Kameny, and E. Lund, "An Overview of the Mermaid System - A Frontend to Heterogeneous Databases," Proc. of EASCON 83, pp. 387-402.

[TEMP 87] M. Templeton, D. Brill, S.K. Dao, E.Lund, P. Ward, A.L.P. Chen, and R. MacGregor, "Mermaid System - A Frontend to Distributed Heterogeneous Databases," Proceedings of the IEEE, Vol. 75, No. 5, (May, 1987), pp.695-707.

[TOER 86] T.J. Teorey, D. Yang, J.R. Fry, "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model," Computing Surveys, Vol. 18, No. 2, (June 1986), pp.197-222.

[TSIC 77] D. Tsichiritzis and A. Klug, eds., "The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Data Base Management Systems," AFIPS Press, Montvale, N.J., (1977).

[ULLM 82] J.D. Ullman, "Principles of Database Systems," second edition, Computer Science Press, Rockville, Maryland, (1982).

[Yu 87] C.T. Yu, K.C. Guh, W. Zhang, M. Templeton, D.Brill, and A.L.P. Chen, "Algorithms to Process Distributed Queries in Fast Local Networks," IEEE Transactions on Computers, Vol. C-36, No. 10, (Oct 1987), pp. 1153-1163.

[ZANI 79] C. Zaniolo, "Design of Relational Views Over Network Schemas," Proc. ACM SIGMOD 79 Conference, Boston, (1979), pp. 179-180.

APPENDIX

GLOBAL QUERY AND MAPPING LANGUAGE SYNTAX

The metalanguage symbols used here have the following meanings:

::=	shall be defined to be
<x>	the non-terminal x
¦	alternatively
.	end of definition
[x]	0 or 1 instance of x
{ x }	0 or more instances of x
(...)	select exactly one of the enclosed alternatives

Terminal symbols are indicated in bold type.

Note that the built-in functions and domains may vary between implementations

<virtual_rel_def> ::= <rel_nm> == <rel_op>.

<query> ::= <rel_nm> == ,rel_op>.

<rel_decl> ::= **base** <rel_nm> **attrs** <attr_decl_lst>;.

<rel_op> ::=
lim <rel_opnd> [**where** <predicate>] [**attrs** <attr_nm_lst>];
¦ **lnj** <rel_opnd>, <rel_opnd> [**where** <predicate>] [**attrs** <attr_nm_lst>];
¦ **onj** <rel_opnd>, <rel_opnd> [**ori** <attr_nm>];
¦ **u** <rel_opnd>, <rel_opnd>;
¦ **int** <rel_opnd>, <rel_opnd>;
¦ **dif** <rel_opnd>, <rel_opnd>;
¦ **div** <rel_opnd>, <rel_opnd>;
¦ **ren** <rel_opnd> **attrs** <attr_nm> **to** <attr_nm> {, <attr_nm> **to** <attr_nm>};
¦ **alt** <rel_opnd> [**drop** <attr_nm_lst>] [**add** <attr_eval_lst>];
¦ **grp** <rel_opnd> **by** <attr_nm_lst> [**add** <attr_eval_lst>];
¦ **trc** <rel_opnd> **row** <attr_nm_lst> **to_col** <attr_nm> **seq** <attr_nm>;
¦ **tcr** <rel_opnd> **col** <attr_nm> **to_row** <attr_nm_lst> **seq** <attr_nm>;.

<rel_opnd> ::= <rel_op> ¦ <rel_nm>.

<attr_nm_lst> ::= <attr_nm> {, <attr_nm>}.

```
<attr eval lst> ::= <attr_eval> {, <attr_eval>}.

<attr eval> ::=
       <attr_decl>
     | <attr_decl> = <function>.

<function> :=
       <expression>
     | <altern_lst>
     | <altern_lst> else <expression>.

<altern lst> ::= <altern> {else <altern>}.

<altern> ::= <expression> if <predicate>.

<attr_decl_lst> ::= <attr_decl> {, <attr_decl>}.

<attr decl> ::= <attr_nm> <domain> | key <attr_nm> <domain>.

<domain> ::=
       numeric <length> [.<scale>]
     | char <length>
     | logical.

<expression> ::=
       <arith_expr>
     | <string_expr>

<arith expr> :=
       (<arith_expr>)
     | <num_lit>
     | <attr_nm>
     | <arith_expr> <bin_ar_op> <arith_expr>
     | - <arith_expr>
     | <arith_fun>
     | <arith_aggr_fun>

<arith fun> ::=
     | (min | max | avg) (<arith_expr>)
     | char_to_num (<string_expr>).

<arith_aggr_fun> ::=
       (agg_avg | agg_sum | agg_max | agg_min)
         (<arith_expr>)
     | count ().

<string expr> ::=
       (<string_expr>)
     | <string_lit>
     | <string_fun>
     | <attr_nm>.

<string_fun> ::= num_to_char (<arith_expr>).
```

```
<predicate> ::=
        (<predicate>)
      ¦ TRUE
      ¦ FALSE
      ¦ <expression> <comparator> <expression>
      ¦ <predicate> <bin_log_op> <predicate>
      ¦ not <predicate>
      ¦ <logical_aggr_fun>.

<logical_aggr_fun> ::= (agg_any ¦ agg_all) (<predicate>).

<comparator> ::= < ¦ > ¦ >= ¦ =< ¦ <>.

<bin_ar_op> ::= + ¦ - ¦ * ¦ / ¦ ^.

<bin_log_op> ::= and ¦ or.

<num_lit> := <digit_string>[.<digit_string>].

<string_lit> ::= "<char_string>" ¦ "".

<digit_string> ::= <digit>{<digit>}.

<digit> := 0 ¦ 1 ¦ 2 ¦ 3 ¦ 4 ¦ 5 ¦ 6 ¦ 7 ¦ 8 ¦ 9.

<char_string> ::= <char>{<char>}.

<char> ::= <any character; " and * are prefixed by *>
```