Extensions to Aldat to Support Distributed Database Operations With No Global Schema

Melanie E. Gaudon
School of Computer Science
McGill University
Montréal, Canada

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

ISBN 0-315-38242-2

ABSTRACT

This thesis introduces a new approach to distributed query processing in which individual hosts are ignorant of the location of remote data. To resolve distributed queries, a host informs its neighbours of what data it is lacking. The neighbours recursively query their neighbours then respond.

This query processing strategy is used to explore extensions to the relational algebra. Both data manipulation and distribution are managed within the framework of the relational model. To achieve this, techniques for the representation and manipulation of queries were developed, along with a canonical representation for data. These new techniques are generally applicable to metadata.

RÉSUMÉ

Ce mémoire présente une nouvelle approche pour le traitement des requêtes dans les bases des données reliées, dans lesquelles un hôte individuel ignore l'emplacement des données à distance. Pour répondre à une requête qui réfère à des données à distance, un hôte informe ses voisins des données qui lui manquent. Ceux-ci transmettent récursivement la requête à leurs voisins avant de répondre.

Cette stratégie de traitement des requêtes est utilisée pour explorer des extensions à l'algèbre relationnelle. La manipulation et la distribution des données sont gérées dans le cadre du modèle relationnel. Pour atteindre ce but, des techniques ont été développées pour la représentation et la manipulation des requêtes, ainsi qu'une représentation canonique des données. Ces nouvelles techniques sont généralement applicables aux métadonnées.

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation for the guidance and encouragement of my supervisor, Dr.T.H.Merrett of McGill University School of Computer Science. While working on this thesis, I would always look forward to our weekly meetings, which were a continued source of interest and motivation.

I am also grateful to Mr.Serge Hurtubise and Mr.Serge LaBelle of Bell Northern Research Limited, and Dr.Robert de B.Johnston of INRS Télécommunications for arranging to finance my work through their organizations.

The staff of BNR and INRS were more than helpful when I required technical assistance and documentation.

Leah Simkin was kind enough to edit and proofread the initial version.

To my friend and husband Paul Gaudon, thank you for everything.

TABLE OF CONTENTS

ABSTRACT .... *i*
RÉSUMÉ .... *ii*
ACKNOWLEDGEMENTS .... *iii*
TABLE OF CONTENTS .... *iv*
LIST OF FIGURES .... *vi*

Chapter 1 Introduction 1
1.1 Thesis Aims and Outline 3
1.2 The Relational Model 4
1.3 Distributed Database Systems 6
1.3.1 SDD-1: A System for Distributed Databases 7
1.3.2 Distributed INGRES 8
1.3.3 R* 9
1.3.4 Other Homogeneous and Heterogeneous DDBMS 10

Chapter 2 Aldat: The Algebraic Data Language 13
2.1 Relations, Data and Metadata 13
2.2 Operations on relations 15
2.2.1 Relational Assignments 16
2.2.2 T-Selectors 17
2.2.3 μ-joins 18
2.2.4 σ-joins 21
2.3 The Domain Algebra 23

Chapter 3 Processing Queries on Data Shared Between Two Hosts 26
3.1 Representing Queries as Relations 27
3.2 Decomposing Queries into Local and Remote Components 30
3.3 The Query Dialogue 34

Chapter 4 Executing Query Relations 42
4.1 A Canonical Representation of Relations 42
4.2 Extensions to Aldat 43
4.3 Transforming Relations Between Representations 46
4.3.1 Converting Relations to Canonical Format 47

4.3.2 Restoring Relations to Their Original Format 51

4.4 Converting Aldat Expressions to and from their Relational Representations 54

4.4.1 Converting Aldat Expressions to QEXP Relations 54

4.4.2 Rebuilding an Aldat Expression from a QEXP Relation 56

Chapter 5 Distributed Query Processing in an Arbitrary Network 58

5.1 Query Propagation 58

5.2 Local Participation in Query Processing 60

5.2.1 Special Purpose Functions and Relations 60

5.2.2 Local Response to an Incoming QEXP Relation 62

5.2.3 Local Response to an Incoming REQUEST Relation 73

5.3 Resolving Distributed Queries 73

Chapter 6 Conclusion 75

6.1 Applications to Metadata and Metacode 75

6.2 Future Work 77

REFERENCES 79

LIST OF FIGURES

1.1 Comparison of the discussed systems. 12

2.1 Four relations represented as tables. 14

2.2 The Aldat assignment operators. 16

2.3 COUNTRIES is obtained by projecting FILMS onto its COUNTRY attribute. 17

2.4 Tuples describing Japanese FILMS are selected to create the JAPAN FILMS relation. 18

2.5 A single T-Selector expression combines a selection operation with a project to produce JAPAN FILMS2. 18

2.6 The natural join of FILMS and CINEMAS. 19

2.7 MALE ACTORS and FEMALE_ACTORS. 20

2.8 The difference join of FILMS and CINEFILM. 20

2.9 The CINEFILM′ and JAPAN_FILMS′ relations are represented in (a). The results of various σ-joins are illustrated in (b),(c), and (d). 22

2.10 An actualization of the virtual attribute PRICE. 24

3.1 The local schemas of hosts A and B. 27

3.2 Query expression 3 1. 28

3.3 Query expression 3.2. 29

3.4 Relational representation of expressions 3.1 and 3.2. 30

3.5 Intermediate results of query decomposition. 33

3.6 Final results of query decomposition. 34

3.7 Local and remote results listed in RESULTS relation. 35

3.8 Query expression 3.3. 36

3.9 QEXP relation with the addition of expression 3.3. 37

3.10 Local decomposition of QEXP relation in figure 3.9. 37

3.11 RESULTS relation (a) originally and (b) after elimination of redundant relations with FINAL_RESULTS and REDUNDANT_RESULTS. 38

3.12 MQEXP is the portion of QEXP combining local and remote results. 39

3.13 DAG representation of MQEXP. 40

3.14 Resulting REQUEST relation to be sent to Host B. 41

4.1 Executing QEXP relations on canonical data. 43

4.2 Relations CINEMAS and CINEFILM with their canonical representation. 44

4.3 Trace of first *while* loop of **Original Canonical** ('CINEMAS'). 49

4.4 Trace of **Original_Canonical** ('CINEMAS') continued to second *while* loop. 50

4.5 Trace of CINEMAS -- **Canonical Original** ('CINEMAS'). 53

5.1 A sample network. 59

5.2 A spanning tree of the sample network. 60

5.3 Metadata relations at Host G. 61

5.4 (a) The incoming QEXP relation. (b) Its graph representation. 64

5.5 New relations created by **Receive_Query** 65

5.6 Trace of **Receive_Results**. (a) Host F answers. (b) RECEIVERS and RESULTS after first iteration. (c) Host H answers. (d) RECEIVERS and RESULTS after second iteration. 67

5.7 Relations created by **Determine Request** with the updated version of RESULTS 68

5.8 Execution of **Request_Relations** on sample problem. (a) The SITES relation (b) Result of initial pick.
(c) REQUEST and TOLIST after first iteration. The REQUEST relation has been sent to Host D.
(d) REQUEST and TOLIST after second iteration. The REQUEST relation has been sent to Host H. 69

5.9 Execution of **Receive Relations** on sample problem. (a) Host H responds with requested data. (b) SITES and CR after first iteration. (c) Host H responds. (d) SITES and CR after second iteration. 70

5.10 LQEXP and RESULTS produced by **Merge**. 71

5.11 FINAL_RESULTS relation produced by **Return Results**. 72

5.12 Trace of **Process Request**. (a) A REQUEST relation arrives from Host D. (b) Before entering loop. (c) After first iteration. (d) After second iteration. 74

6.1 An integrity constraint represented by a QEXP relation. 76

6.2 PERSON is a generalization of ACTOR and DIRECTOR. 77

Chapter 1 — Introduction

The database approach evolved from the need for centralized control over information resources. The initial tendency was to assemble all data at one location where a single database administrator would assure data integrity and security. All applications stored and retrieved data from the central database. This implicitly hierarchical structure of centralized databases conflicts with the decentralized nature of many organizations. For example, a manufacturing company with plants in various cities across the country may have a small computing center at each location maintaining personnel and inventory files. The local files would be used at their location of origin to process pay checks and keep track of stock. Only occasionally, such as when the head office is drawing up a balance sheet or comparing plant productivity, is it necessary to have a global view of data from all plants.

The local files could be incorporated into a centralized database, however this solution has its drawbacks: data would most often be stored away from its point of origin; local applications would have to access the central database remotely, entailing high communications costs. Moreover, in case of failure at the central database location, all applications would be suspended. Such shortcomings led to the development of distributed databases.

A distributed database system allows each site to carry out applications on its local data independently, while providing a global view of data when needed. As a centralized database expands it will eventually reach capacity and may have to be replaced to allow further growth. Distributed databases may grow incrementally with the addition of new, relatively autonomous units, thus minimizing the impact of expansion on the existing system.

We define the database schema as the catalogue of available data. In a distributed database, a query may reference locally accessible data as well as data from a remote site. At each site in a distributed database, a local schema should describe locally available data. This assures that applications involving only local data can execute independently of any central control. Queries referencing remote data need schema information not available from the local schema. The question of how they may obtain this information is a major design issue in distributed database management systems (DDBMS).

One solution is to replicate a global schema at each host. However, if the local host is a workstation accessing a very large distributed database, the global schema may be too large to store locally. Also, when the schema is replicated, any changes to it necessitate a global update of all local copies. An alternative to having a copy of the global schema at each site is to designate one site to maintain the global schema; all applications access the global schema via this location. This solution eliminates many of the advantages of distributing the database; a failure at the schema's site cripples all processing and repeated accessing of the global schema creates a system bottleneck. A compromise between the above two strategies could also be considered. In existing distributed database systems, to our knowledge, all hosts maintain some global schema information locally to access remote data.

1.1 Thesis Aims and Outline

In this thesis we propose a dialogue model for distributed query processing in which independent processors can cooperate to resolve distributed queries with no a priori knowledge of the global schema. We attempt to manage both data processing and distribution within the framework of the relational model. The strategy will be illustrated using the Algebraic data language, Aldat [MER 84]

The next two sections of this chapter introduce the relational model and give an overview of distributed database research. The research prototypes described are meant to be contrasted with the dialogue model presented in this work.

Chapter 2 defines relations and introduces the subset of the Aldat language that will be used in subsequent chapters. Illustrative examples rather than formal definitions will be emphasized.

In Chapter 3 we present the dialogue model in a network restricted to two hosts. A relational representation of queries is proposed which can be factored into local and remote components. We show how a host lacking data to complete a query, can engage in a dialogue with its neighbour to locate and obtain the necessary data.

A general execution program is discussed in chapter 4 to execute query relations on a canonical data representation. Some extensions to Aldat are suggested to support conversion between different representations of queries and data. The conversion algorithms illustrate the usefulness of the extensions.

The dialogue model is extended to an arbitrary network in Chapter 5. Chapter 6 concludes the thesis and suggests topics for further research.

1.2 The Relational Model

First proposed as a database model by Codd in 1970 [COD 70], the relational model has become the basis of virtually all current database research. In 1979 Kim [KIM 79] published a survey of Relational Database Management Systems (DBMS). Fernandez [FERN 80] added to this list of existing systems in 1980. Together, they enumerate over thirty different systems. With the advent of powerful micro computers, numerous Relational DBMS have been implemented for personal computers. In 1983 a two day workshop organized by the INRIA, France, was devoted to design and implementation of Relational DBMS on micro-computers [INR 83].

The success of the relational model can be attributed to its simplicity: all information is represented by a single data structure, the relation. With the relational model, as opposed to previous database models, details of implementation such as access paths and indexing may be ignored, as they do not affect the programmers' view of data. This property is referred to as *physical data independence*.

From its conception, the relational model provided algebraic operators to manipulate relations [COD 70]. The relational algebra operators accept one or two relations and produce a new relation as a result. This property of *closure* allows nested relational expressions: expressions in which operands are themselves expressions. As relations are manipulated as entities, the relational algebra is less procedural than data manipulation languages requiring record by record loops. The relational algebra is discussed in more detail in Chapter 2.

Another formalism for manipulating relations, the relational calculus [COD 72], was proposed as even less procedural than the relational algebra. While the relational algebra provides a set of operators that may be applied to achieve a final result, the relational calculus describes the desired result; it is left up to the system to determine the necessary operations. The relational calculus and algebra are fundamentally equivalent as any expression in one may be translated into an equivalent expression in the other; for proof see [COD 72] and [ULL 82]. Experiments conducted on the ability of programmers to write complex queries in languages differing primarily in their procedurality [WEL 81], have shown a higher success rate among those using the more procedural language. A procedural language allows a programmer to decompose a complex problem into a series of simpler tasks. As Aldat is meant to deal with the programming aspects of relational database, the relational algebra has been adopted as the more suitable language model.

In light of its enormous success in the area of administrative data processing, researchers are extending the relational model to non-conventional applications. For instance, the PROBE project [DAY 85] is developing a small set of useful extensions to relational database management systems to support knowledge representation, processing of spatial and temporal data, and the representation and manipulation of complex objects. The PROBE architecture provides an extendible DBMS that may interact with specialized hardware and software such as image processors or keyword-based information retrieval systems. Korth [KOR 86] incorporates new features into the relational model that include aspects of object oriented and functional programming. These extensions have been applied to the relational operating system interface, ROSI, which models all objects in a user's environment as relations [KOR 86].

1.3 Distributed Database Systems

The implementation of distributed database systems adds the problems of computing networks to those inherent to database systems. Aspects of database management, such as concurrency and integrity control, become even more complex once the database is distributed. Distributed database systems may be homogenous or heterogeneous: homogeneous systems integrate sites running the same DBMS, while heterogeneous systems incorporate various processors running different DBMS software. Systems also differ with respect to site autonomy. Most use the relational model as its simplicity facilitates data distribution

To merit the appellation *distributed*, a DBMS should provide distribution transparency, i.e., the user should not be concerned with the actual location of the data He accesses the database as if it were entirely local. The elimination of redundancy is a major motivation in centralized databases, however, redundancy is often desirable in a distributed database. Multiple copies of data provide backups in case one copy is destroyed. Performance of read-only applications may be enhanced by allowing a processor to choose the closest of several copies. In the event of site failure, alternative sources of data are available. Redundancy complicates update applications as all copies must be modified.

An overview of commercial products providing aspects of distributed database management is presented in [CER 84]. Many more sophisticated systems are under development. The following paragraphs briefly describe a representative sample of DDBMS research prototypes. The discussion will focus on locating remote data and site autonomy.

1.3.1 SDD-1: A System for Distributed Databases

Developed at the Computer Corporation of America, SDD-1 was the first distributed DBMS. Reference [ROT 80] introduces the general principles. The query processing strategy is described in [BER 81.b]. SDD-1 supports the relational data model. Users interact with the database through a high-level procedural language, Datalanguage. Relations may be fragmented according to the rules outlined in [ROT 80] and fragments may be replicated. The user has a global view of the entire database and is not affected by replication.

The schema of an SDD-1 database is provided by directory relations. Directories are treated as any user data and may therefore be replicated and fragmented. A *directory locator* stored redundantly at each site serves as a global schema by indicating where each directory fragment is stored.

The SDD-1 architecture is based on three relatively autonomous virtual machines: data modules (DM), transaction modules (TM) and the reliable network (RelNet). The DMs read, write and manipulate data in local workspaces. They can also move data from one local workspace to another. Each individual Datalanguage statement composes a separate transaction. The TMs translate the query, establish an access plan, control concurrency and query execution. The RelNet provides guaranteed delivery of messages, controls transactions, and monitors the network.

Transactions are executed in three phases: read, execute, and write. During the read phase, the TM at the site of origin uses the *directory locator* to determine what data will be accessed and where it is located. It then broadcasts a command to the concerned sites to put the relevant data in local workspaces assigned to the transaction. The TM, at the site of origin controls the local DMs during the execution phase. The write phase updates all copies of modified fragments and guarantees database integrity.

1.3.2 Distributed INGRES

Distributed INGRES was developed at the University of California, Berkely, as an extension to the previously operative DBMS, INGRES [STO 76]. The user interacts with a global view of the different databases in the network. Data may be replicated and fragmented. As in SDD-1, the site at which the query originates controls the remote processes involved in the query processing.

The INGRES system catalogue contains four types of information [STO 76].

1. Relation names and locations
2. Parsing information (domain names, format, etc.)
3. Performance information (number of tuples, storage structure, etc.)
4. Consistency information (protection, integrity constraints etc.)

The database contains two classes of relations: local, which are only accessible from the host at which they are located, and global, which are accessible throughout the database. Each site keeps systems catalogues for its local relations and global relations residing at its location. Type 1 catalogue information for all global relations is stored redundantly at each site. This constitutes a global schema. A new global relation is created by broadcasting its name and location to all sites in the network

Before query processing, types 2, 3, and 4 of catalogue information must be assembled at the site where the query originates. The assembled data is saved locally as a working copy so that subsequent transactions need not request the information again. Working copies are not updated, and after a certain period of time they are considered out of date and discarded. This strategy works well if the catalogue is relatively static; however, run-time errors may occur because of inaccurate information in the working catalogue.

1.3.3 R*

The R* project [WIL 82] is the distributed version of the IBM System R relational database system. The commercial database product SQL/DS is an extension of System R The R* system is intended to provide most of the features of SQL/DS in a distributed environment. Site autonomy was a major consideration in the development of R*; each site controls access to its own data, and local data is manipulated with a minimum of interference from other sites. The addition of new sites to the network does not alter global data structures or definitions. Some replication and fragmentation of relations is supported.

The global schema is provided by *system wide names* (SWN). Users reference data objects by local *print names* that are mapped onto corresponding SWNs by the system. The SWN specifies the relation's creator, the creator's site, the relation's name, and initial location or birthsite. A relation's actual location is dynamic and does not appear in its name. Whenever a relation moves from one site to another its new location is indicated at its birthsite. Thus the SWN allows any relation to be accessed in at most two attempts. If the relation is not present at its birthsite, the birthsite indicates its current location.

The site at which a query originates, referred to as the master, determines the system wide names of all relations involved in a query and produces a global compilation of the query. All schema information is assembled at the master prior to processing. Remote hosts involved in query processing are called apprentices. The full global execution plan is distributed to all apprentices who may determine the best strategy for manipulating their local data.

1.3.4 Other Homogeneous and Heterogeneous DDBMS

Descriptions of other DDBMS prototypes may be found in references [CER 84], [DEL 80], [PAG 85], [SUZ 82] and [AND 82]. The Genesis system [PAG 85] is of particular interest as it is based on a distributed operating system. The database system is thus relieved of responsibility for locating remote data, as the operating system provides a global view. Tasks such as creation and control of remote task execution, synchronization, recovery, etc., may be implemented more efficiently at the operating system level. The commercial Tandem/ENCOMPASS system for distributed databases [BOR 81] is also based on the concept of a distributed operating system.

When a distributed database is to be implemented from scratch, it makes sense to have the same DBMS software at each site. However often it is desirable to integrate several existing databases with different DBMSs and perhaps even different data models. SIRIUS-DELTA [FERR 82] and Multibase [SMI 81] are both heterogeneous DDBMS prototypes.

SIRIUS-DELTA is part of a French nationwide project on distributed databases. A minimal set of functions called the *pivot system* is defined as a common reference. Each DBMS request is mapped into an equivalent request in the *pivot system*. The system architecture has two levels: global and local. The global level maintains a global internal schema specifying how to rebuild global data from local data. The global levels acts as a user of the local DBMSs.

Multibase is a Computer Corporation of America project which integrates pre-existing, heterogeneous distributed databases. Each database has its own *local host schema* (LHS). These schemata can correspond to different DBMS, data models, and query languages. Each LHS is mapped onto a *local schema* (LS) corresponding to the Functional Data Model [SHI 79]. This provides a homogeneous view of the different systems. An *integration schema* (IS) describes a database containing information about mapping the inconsistent data models into a *global schema* (GS). The IS and LSs are combined to define the GS. The GS is queried via the DAPLEX language [SHI 79].

	Remote Data Located Via:	Site Autonomy 1: high, 2: medium, 3: low	Replication and Fragmentation Transparency
SDD-1	Directory locator	3	yes
Distributed INGRES	Type 1 catalogue information	3	yes
R*	System Wide Names	1	yes
Genesis	Operating System	3	yes
SIRIUS-DELTA	Global Internal Schema	2	yes
Multibase	Global and Integration Schemata	2	yes

Fig. 1.1 Comparison of the discussed systems.

The table in figure 1.1 recapitulates the features of the different systems we have discussed. The methods by which remote data is located and accessed are summarized in the first column. All systems use a global schema of some sort. The second column indicates the level of site autonomy. The R* system demonstrates the most site autonomy as local sites have control over access to their own data. The other three homogeneous systems have limited site autonomy, as one process at the site at which a query originates controls slave processes at the other sites participating in the transaction. Individual sites in the heterogeneous systems can have some control over the mapping of their local data to the global schema. All of these systems support some form of replication and fragmentation transparency.

Chapter 2 Aldat: The Algebraic Data Language

The Aldat project, headed by Dr.T.H.Merrett at McGill University, explores extensions and applications of the relational algebra. The extensions have evolved through a slow empirical process of developing applications of the existing formalism and extending only where necessary and only if the extension fits into a simple conceptual framework. The basis of Aldat is described in [MER 84]. After defining relations, this chapter outlines the subset of Aldat used in this thesis. The syntax used is that of [MER 84] with the exception that recursive assignment statements will be allowed. In following chapters we adopt Pascal-like constructs such as function and procedure headings, variable declarations, *if* statements and *while* loops. This allows us to write Aldat routines that interpret other Aldat statements. These functions and procedures will be referred to as *metacode* routines.

2.1 Relations, Data and Metadata

Figure 2.1 represents four instances of relations containing data about various films and the cinemas showing them.

FILMS (TITLE YEAR DIRECTOR COUNTRY)

TITLE	YEAR	DIRECTOR	COUNTRY
Arigato-san	1936	H.Shimizu	Japan
L'Avare	1980	L.de Funes	France
Carefree	1938	M.Sandrich	U.S.A
Hana Saku Minato	1943	K.Kinoshita	Japan
Miss Julie	1941	A.Sjoberg	Sweden
Henry V	1944	L.Olivier	G.B.
Medea	1970	P.P.Pasolini	Italy
Swing Time	1936	G.Stevens	U.S.A.
Rembetiko	1983	C.Feris	Greece

ACTORS (ACTOR TITLE)

ACTOR	TITLE
F.Astaire	Swing Time
G.Rogers	Swing Time
K.Uehara	Arigato-san
E.Ozawa	Hana Saku Minato
K.Uehara	Hana Saku Minato
S.Leonardou	Rembetiko
F.Astaire	Carefree
G.Rogers	Carefree
M.Callas	Medea
L.de Funes	L'Avare
M.Galabru	L'Avare
L.Olivier	Henry V
R.Newton	Henry V
A.Bjork	Miss Julie
U.Palme	Miss Julie

CINEFILM (TITLE CINEMA SCREEN)

TITLE	CINEMA	SCREEN
Swing Time	Roxy	1
Rembetiko	Roxy	2
L'Avare	Odeon	1
Hana Saku Minato	Bijou	1
Arigato-san	Bijou	2
Medea	Bijou	3
Arigato-san	Capital	1

CINEMAS (CINEMA ADDRESS SCREENS)

CINEMA	ADDRESS	SCREENS
Roxy	123 Main St.	2
Odeon	987 Park Ave.	1
Bijou	645 Maple Rd.	3
Capital	392 Union Blvd.	1

Fig. 2.1 Four relations represented as tables.

When relations are represented as tables, rows describe objects and columns correspond to properties of the objects. In all the relations we will consider, a single value is associated with each row/column intersection. Relations satisfying this criterion are said to be *normalized*. The relation's rows are called *tuples*, hence the definition of a relation as a set of tuples. The columns are called *attributes*. Each attribute has a corresponding *domain*, a non-empty set from which it takes its values. Each tuple is unique within a relation and the order of tuples is completely arbitrary, just as elements in a set are unique and not ordered. As the attributes are all labelled, their order is also arbitrary. In other words, any permutation of rows and columns in the tabular representation does not in any way alter the information contained in the relation.

A finite set of attribute names $\{A_1, A_2, ..., A_n\}$ is called a relational scheme [MAI 83]. Each attribute has a corresponding set D_i, its domain. We will formally define a relation on the above relational scheme as a subset of the extended cartesian product $D_1 \times D_2 \times ... \times D_n$ [MER 84].

A relational database is comprised of data and metadata. Data is represented by relations such as those in figure 2.1. Metadata is data that describes or helps to interpret other data [DAY 85]. In a relational database metadata includes: the names of relations and their corresponding attributes; types and domains of attributes; physical storage and access paths for relations; etc. Metadata may also be represented as relations. In our metacode routines, metadata is accessed as any other data. Not all environments allow users to access metadata.

2.2 Operations on relations

While relations offer a means of representing data, the relational algebra provides a means of manipulating data. Aldat is an extension of Codd's relational algebra first proposed in [COD 70]. The unary operators, *select* and *project* become *T-Selectors* (tuple selectors) in Aldat. As relations are generalizations of sets, Aldat has relational operations which are generalizations of set operations. The set-valued set operations (union, intersection, etc.) become the class of *μ-joins*. Logic-valued set operations (inclusion, empty intersection, etc.) are extended to the class of *σ-joins*. The natural join is a special case of μ-joins. Natural composition and relational division are special cases of σ-joins.

2.2.1 Relational Assignments

Aldat has two relational assignment operators. The expression, $T \leftarrow R$, replaces relation T with relation R. The incremental operator, $T \leftarrow\!+\, R$, appends the tuples of R to T. Figure 2.2 shows the results of several different assignment operations on sample relations.

INITIAL VALUES

T(B C A)	R(A B C)	S(A D E)
q r s	a b c	x y z

ASSIGNMENT	RESULT T(B C A)
T <-- R	b c a
T [B, C, A <-- A, E, D] S	x z y
T <--+ R	q r s b c a
T [B, C, A <--+ A, D, E] S	q r s x y z
T [B, C, A <--+ B, C, 'w'] R	q r s b c w

Fig. 2.2 The Aldat assignment operators.

When attribute names are not specified in the assignment statement, the attributes of the operand are implicitly assigned to attributes with the same name in the resulting relation. The correspondence between attributes in the operand relation and attributes in the resulting relation may also be indicated explicitly.. Care must be taken not to assign an attribute to another with a conflicting domain. For instance, a character valued attribute may not be assigned to an integer valued attribute.

2.2.2 T-Selectors

Projection operations specify a subset of a relation's attributes. A relation containing only the countries present in the FILMS relation of figure 2.1 may be obtained by projecting FILMS onto its attribute COUNTRY. This operation is expressed by:

COUNTRIES — COUNTRY in FILMS ;

The resulting relation is shown in figure 2.3. Note that FILMS has nine tuples and COUNTRIES only has seven. This is because the values "Japan" and "USA" appear twice in the attribute COUNTRY of FILMS. The projection operation eliminates duplicate tuples in the resulting relation.

COUNTRIES (COUNTRY)
Japan
France
U.S.A
Sweden
G.B.
Italy.
Greece

Fig. 2.3 COUNTRIES is obtained by projecting FILMS onto its COUNTRY attribute.

Selection operators extract a subset of a relation's tuples satisfying a given criteria. The tuples of FILMS pertaining to Japanese films may be selected into a relation JAPAN_FILMS by the following select operation.

JAPAN_FILMS (	TITLE	YEAR	DIRECTOR	COUNTRY)
	Arigato-san	1936	H.Shimizu	Japan
	Hana Saku Minato	1943	K.Kinoshita	Japan

Fig. 2.4 Tuples describing Japanese FILMS are selected to create the JAPAN_FILMS relation.

JAPAN_FILMS ← where COUNTRY = 'Japan' in FILMS ;

Figure 2.4 shows the result of this operation.

Aldat allows project and select operations to be combined in single T-Selector expressions. For instance, if one wishes to retrieve only the titles and directors of Japanese films, the expression:

JAPAN_FILMS2 ← TITLE, DIRECTOR where COUNTRY = 'Japan' in FILMS ;

creates the relation shown in figure 2.5.

JAPAN_FILMS2 (	TITLE	DIRECTOR)
	Arigato-san	H.Shimizu
	Hana Saku Minato	K.Kinoshita

Fig. 2.5 A single T-Selector expression combines a selection operation with a project to produce JAPAN_FILMS2.

2.2.3 μ-joins

The *natural join* is the most common member of the μ-join family [COD 70]. The relation SHOWING in figure 2.6 is the natural join of relations FILMS and CINEFILM. The natural join associates tuples of FILMS with tuples of CINEFILM having the same value of TITLE, their common attribute. The values of TITLE

SHOWING (& TITLE	YEAR	DIRECTOR	COUNTRY	CINEMA	SCREEN)
Arigato-san	1936	H.Shimizu	Japan	Bijou	2
L'Avare	1980	L.de Funes	France	Odeon	1
Hana-Saku Minato	1943	K.Kinoshita	Japan	Bijou	1
Medea	1970	P.P.Pasolini	Italy	Bijou	3
Swing Time	1936	G.Stevens	U.S.A.	Roxy	1
Rembetiko	1983	C.Feris	Greece	Roxy	2
Arigato-san	1936	H.Shimizu	Japan	Capital	1

Fig. 2.6 The natural join of FILMS and CINEMAS.

appearing in SHOWING are the intersection of TITLE in FILMS with TITLE in CINEFILM.

In Aldat the natural join is called the intersection join and is designated by the *ijoin* operator. The expression:

$$SHOWING \leftarrow FILMS \text{ ijoin } CINEFILM \, ;$$

produces the result in figure 2.6. SHOWING would satisfy a request to know which films are currently showing at some cinema. The attributes participating in the join (in this example, the attribute TITLE) are called the join attributes. As with assignment operators, join attributes may be specified implicitly or explicitly. The natural join of relations on none of their attributes computes their cartesian product.

Two other μ-joins, corresponding to set union and difference, will also be used in our metacode routines. Imagine that the men and women starring in the films are listed separately in the relations MALE_ACTORS and FEMALE_ACTORS of figure 2.7.

The ACTORS relation of figure 2.1 could be reconstructed by the following union join.

$$ACTORS \leftarrow MALE\_ACTORS \text{ ujoin } FEMALE\_ACTORS \, ;$$

MALE_ACTORS(ACTOR	TITLE)
F.Astaire	Swing Time
K.Uehara	Arigato-san
E.Ozawa	Hana Saku Minato
K.Uehara	Hana Saku Minato
F.Astaire	Carefree
L.de Funes	L'Avare
M.Galabru	L'Avare
L.Olivier	Henry V
R.Newton	Henry V
U.Palme	Miss Julie

FEMALE_ACTORS(ACTOR	TITLE)
G.Rogers	Swing Time
S.Leonardou	Rembetiko
G.Rogers	Carefree
M.Callas	Medea
A.Bjork	Miss Julie

Fig. 2.7 MALE_ACTORS and FEMALE ACTORS.

The set of tuples in ACTORS is the union of the operand relations.

The difference join selects tuples of the left operand that do not participate in the natural join. It is actually the left difference join, however, since we do not use the right difference join, we will simply refer to it as the difference join The difference join of FILMS with CINEFILM, shown in figure 2.8, is expressed as:

$$NOTSHOWING \leftarrow FILMS\ djoin\ CINEFILM\ .$$

It may be interpreted as the films that are not showing at any cinema.

NOTSHOWING (TITLE	YEAR	DIRECTOR	COUNTRY)
Carefree	1938	M.Sandrich	U.S.A
Miss Julie	1941	A Sjoberg	Sweden
Henry V	1944	L.Olivier	G.B.

Fig. 2.8 The difference join of FILMS and CINEFILM.

The join attribute is once again TITLE. The set of values in the TITLE attribute of NOT_SHOWING is the difference between the set of values of TITLE in FILMS and the set of values of TITLE in CINEFILM.

The family of μ-joins is defined in [MER 84] in terms of three disjoint sets of tuples. For given operand relations, R(X,Y), S(Y,Z), these sets are defined on the attributes (or attribute groups) X, Y, Z as:

$$\mathit{center} \stackrel{\triangle}{=} R \ \textbf{ijoin} \ S$$

$$\mathit{left\ wing} \stackrel{\triangle}{=} \{(x,y,DC) | (x,y) \in R \ \textbf{and} \ \forall z((y,z) \notin S)\}$$

$$\mathit{right\ wing} \stackrel{\triangle}{=} \{(DC,y,z) | (y,z) \in S \ \textbf{and} \ \forall x((x,y) \notin R\}.$$

The tuples from R that match no tuples of S, augmented by DC, form the *left wing*. Similarly, the tuples from S that match no tuples of R, augmented by DC, form the *right wing*. DC is a null value meaning *Don't Care* [MER 84]. The union and difference joins may now be defined as:

$$R \ \textbf{ujoin} \ S \stackrel{\triangle}{=} \mathit{left\ wing} \cup \mathit{center} \cup \mathit{rightwing}$$

$$R \ \textbf{djoin} \ S \stackrel{\triangle}{=} X,Y \text{ in } \mathit{left\ wing}.$$

2.2.4 σ-joins

The family of σ-join operators are also called *set selectors*. They accept a value of an attribute (or group of attributes) if the set of tuples associated with it satisfies a specified condition. The condition is a set comparison. The sets are the operand relations. We will illustrate four σ-join operations on the relations; CINEFILM′ and JAPAN_FILMS′ of figure 2.9.a. These are the set subset selector, the set inclusion selector, the null intersection set selector, and natural composition.

CINEFILM′ is the projection of CINEFILM onto its CINEMA and TITLE attributes. JAPAN_FILMS′ is JAPAN_FILMS projected onto TITLE. Each value of the CINEMA attribute in CINEFILM′ is associated with a set of values of the TITLE attribute. These sets are labeled 1 to 4 in the illustration.

Suppose one wishes to select the cinemas showing all of the Japanese films in our database. The JAPAN_FILMS$'$ relation is the set of titles of all Japanese films. We need to select the cinemas such that the set of titles of the films they are showing contains the set of titles of all Japanese films. These values will be selected by the set subset selector in the following expression.

$$ALL\_JAPAN \leftarrow JAPAN\_FILMS' \subseteq CINEFILM' ,$$

The set inclusion selector will return the same result in the expression;

$$ALL\_JAPAN \leftarrow CINEFILM' \supseteq JAPAN\_FILMS' ;$$

The set inclusion selector is equivalent to relational division [COD 72]. ALL_JAPAN is shown in figure 2.9.b.

CINEFILM' (CINEMA TITLE)

CINEMA	TITLE	
Roxy	Swing Time	1
Roxy	Rembetiko	
Odeon	L'Avare	2
Bijou	Hana Saku Minato	
Bijou	Arigato-san	3
Bijou	Medea	
Capital	Arigato-san	4

JAPAN_FILMS' (TITLE)

TITLE
Arigato-san
Hana Saku Minato

(a)

ALL_JAPAN (CINEMA)

CINEMA
Bijou

(b)

NO_JAPAN (CINEMA)

CINEMA
Roxy
Odeon

(c)

SOME_JAPAN (CINEMA)

CINEMA
Bijou
Capital

(d)

Fig. 2.9 The CINEFILM$'$ and JAPAN_FILMS$'$ relations are represented in (a). The results of various σ-joins are illustrated in (b),(c), and (d).

Our user now wishes to select cinemas showing no Japanese films. The set of titles associated with these cinemas will not intersect with the titles in the JAPAN_FILMS$'$ relation. The following null intersection set selector will return the desired values.

$$NO\_JAPAN \leftarrow CINEFILM' \cap JAPAN\_FILMS' .$$

The result of this operation can be viewed in figure 2.9.c.

The *natural composition* operation is the contrary of the null intersection set selector. Natural composition selects values if their corresponding sets have a non-empty intersection with the reference set. Natural composition is equivalent to a natural join followed by a projection onto all attributes except those participating in the join. In our sample database; one can select the cinemas playing some Japanese films with the following natural composition.

$$SOME\_JAPAN \leftarrow CINEFILM' \text{ icomp } JAPAN\_FILMS' ;$$

SOME JAPAN is illustrated in figure 2.9.d

2.3 The Domain Algebra

The domain algebra defines *virtual attributes* which may be *actualized* when needed. When utilized to its full potential, the domain algebra provides facilities such as arithmetic, totaling, ordering, etc. A thorough description of the domain algebra may be found in [MER 84]. In this thesis we will use the domain algebra to assign constant values to virtual attributes and to rename actual attributes.

Suppose that in our sample database all cinemas charge the same admission price to all films We can define a virtual attribute as follows:

let PRICE be $5.50 .

Virtual attributes are not associated with any relation until actualized, usually by projection. Virtual attributes must be actualized before being used by the relational algebra. The following expression will produce the result in figure 2.10.

CINEPRICE(CINEMA	ADDRESS	PRICE)
Roxy	123 Main St.	$5.50
Odeon	987 Park Ave.	$5.50
Bijou	645 Maple Rd.	$5.50
Capital	392 Union Blvd.	$5.50

Fig. 2.10 An actualization of the virtual attribute PRICE.

CINEPRICE - - CINEMA, ADDRESS, PRICE in CINEMAS ;

In this simple example in which the virtual attribute is defined as a constant value, PRICE could be replaced by a singleton unary relation, PRICE, with the attribute PRICE having the value $5.50. The result in figure 2.10 could have been obtained by performing the cartesian product of the relation PRICE with the projection of CINEMAS onto CINEMA and ADDRESS. In more complex expressions virtual attributes may not be replaced by relations as their values are determined by actualization. The necessary distinction between virtual attributes and relations will become more apparent in chapter 4.

PRICE can be actualized on any relation in the database. One meaningful example would be to project the relation FILMS onto its actual attributes as well as PRICE to include the admission price in the representation. It would not be meaningful to actualize price on the ACTORS relation.

Attributes may be renamed by defining an equivalent virtual attribute. The Aldat statement:

let NATIONALITY be COUNTRY ;

renames the COUNTRY attribute. A reference to the virtual attribute NATIONALITY refers to the actual attribute COUNTRY. The utility of these constructs will be illustrated in the conversion routines of Chapter 4.

Chapter 3 Processing Queries on Data Shared Between Two Hosts

We will now distribute the sample database between two processors, Host A and Host B. The relations will neither be fragmented nor replicated The data distribution must be transparent to the user. Therefore, it is up to the DBMS, in our case Aldat, to determine what portion of the query may be resolved locally and what portion requires remote data The hosts only have knowledge of their local schemas. They communicate by exchanging messages in the form of relations.

The query execution algorithm is:

1. Decompose into local and remote subqueries.
2. In parallel :
   - 2.1 Execute local query.
   - 2.2 Transmit remote query and receive list of remotely available data.
3. Determine which remotely available data are needed to complete original query and request that they be sent by neighbour.
4. Once remote results have been received, merge with local results to obtain the answer to original query.

In the discussion of our distributed query processing strategy, we assume that the network is reliable, i.e. all messages are received without error. Optimization has been neglected as we wish to present a feasible strategy before trying to optimize. We will however mention it briefly in chapter 6. The database is assumed to be consistent, and as queries are read-only it will remain so.

At each host there is a metadata relation, SCHEMA, describing locally accessible data. The SCHEMA relation lists all local relations in its RNAME field with their corresponding attributes in the DNAME field. Figure 3.1 shows how the database of Chapter 2 could be distributed between the two hosts.

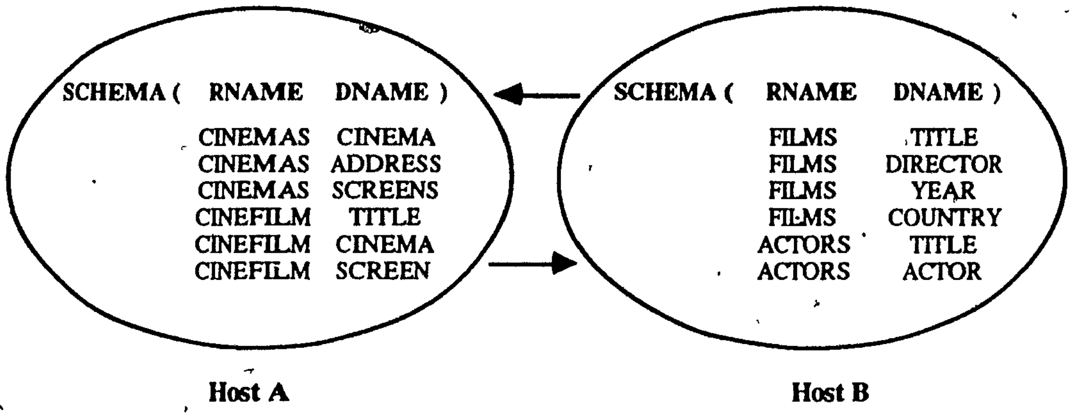


Fig. 3.1 The local schemas of hosts A and B.

The remote subquery may be considered as the original query minus the local schema. As the schema is represented as a relation, we will also represent the query as a relation. The schema may then be subtracted from the query using Aldat operators.

3.1 Representing Queries as Relations

Consider the following query on the database of Chapter 2 and its Aldat representation :

"Find the names and addresses of cinemas showing films starring Lee Marvin."

LEE_FILMS ← *CINEMA, ADDRESS in* 3.1
((where ACTOR = 'Lee Marvin' in ACTORS ijoin CINEFILM)
ijoin (CINEMAS) ;

Figure 3.2 represents the Aldat expression as a tree.

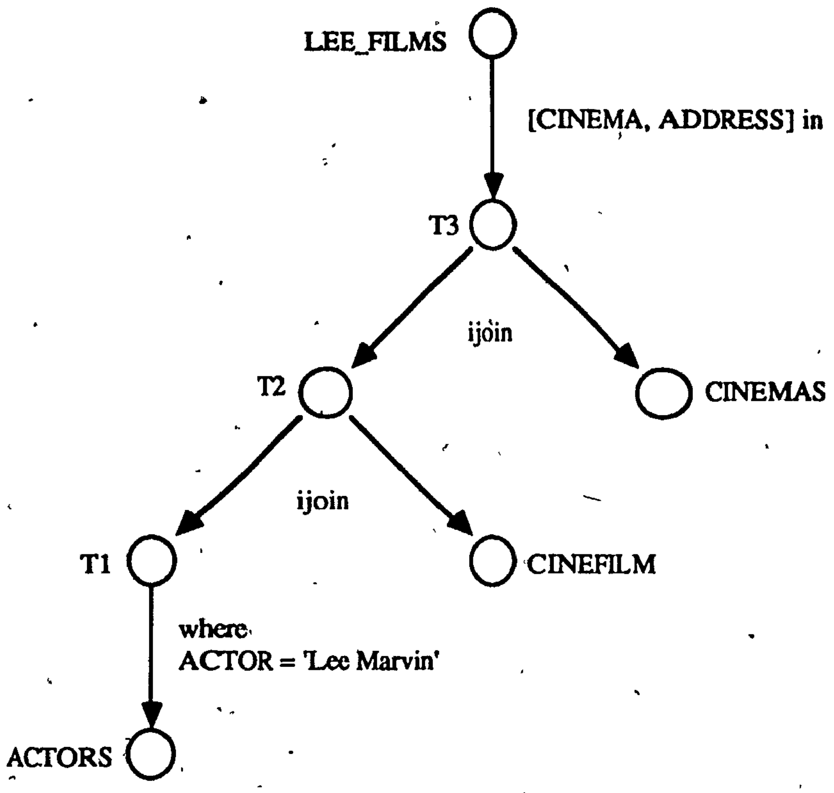


Fig. 3.2 Query expression 3.1.

The root node corresponds to the result, leaves are the operands. Other nodes are temporary relations containing intermediate results. A query is not necessarily a tree but rather a directed acyclic graph (DAG). The expression:

"Find the films starring Fred Astaire but not Ginger Rogers."

FRED_ONLY ← *(TITLE where ACTOR = 'Fred Astaire' in ACTORS)* 3.2
djoin (TITLE where ACTOR = 'Ginger Rogers' in ACTORS) ;

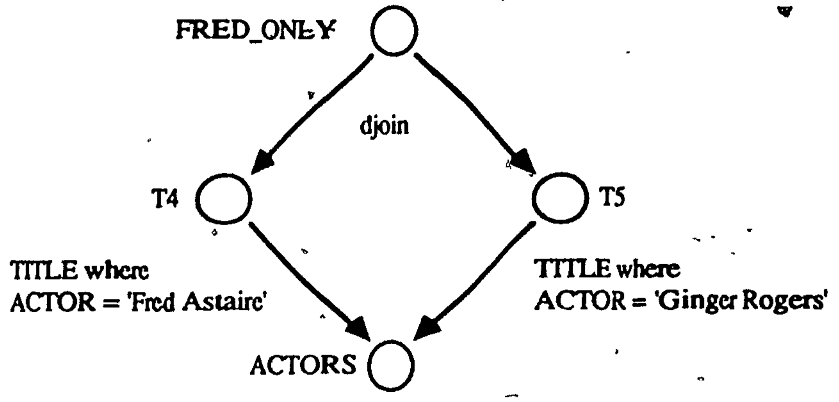


Fig. 3.3 Query expression 3.2.

is represented in figure 3.3.

A directed graph may be represented as a binary relation with a tuple for each edge. Attributes correspond to the source and destination nodes. Additional attributes are needed to completely describe a query. The QEXP relation in figure 3.4 represents expressions 3.1 and 3.2.

The RES and RNAME attributes are the source and destination nodes of an edge. Destination nodes are in fact the operands of a relational operation resulting in a common source node. An operand may be a permanent data relation or the result of previous operations. An OP attribute is required to specify which operation produces the result in RES. ORD identifies the operand in RNAME as left (L) or right (R) for binary operations, or as unique (U) for unary operations. Finally, TYPE indicates whether RES is a resulting (R), temporary (T), permanent (P) or available (A) relation. Initially, only the first three values appear in TYPE.

Each edge of the expression DAG has a corresponding tuple. While writing code to manipulate QEXP, it was found desirable to have an entry for each node in the RES

QEXP(RES	OP	RNAME	ORD	TYPE)
LEE_FILMS	[CINEMA, ADDRESS]	T3	U	R
T3	ijoin	T2	L	T
T3	ijoin	CINEMAS	R	T
T2	ijoin	T1	L	T
T2	ijoin	CINEFILM	R	T
T1	where ACTOR='Lee Marvin'	ACTORS	U	T
ACTORS	id	ACTORS	U	P
CINEFILM	id	CINEFILM	U	P
CINEMAS	id	CINEMAS	U	P
FRED_ONLY	djoin	T4	L	R
FRED_ONLY	djoin	T5	R	R
T4	TITLE where ACTOR='Fred Astaire'	ACTORS	U	T
T5	TITLE where ACTOR='Ginger Rogers'	ACTORS	U	T

Fig. 3.4 Relational representation of expressions 3.1 and 3.2.

field. To achieve this, a tuple for each terminal node is added to QEXP with common values for both RES and RNAME resulting from the identity (*id*) operator. After this addition, QEXP, is no longer a true DAG as terminal nodes are unit cycles. We choose to ignore this discrepancy and continue referring to the graph representations of expressions as DAGs.

Several relations of the QEXP format may participate in a given application, each containing one or many expressions. In our examples, names of QEXP relations always terminate with "QEXP" to distinguish them easily from data relations.

3.2 Decomposing Queries into Local and Remote Components

As was stated in section 3.1, the terminal nodes of the expression DAG are the operand relations. An operand is local if it is present in the local schema. A result relation, either intermediate or final, can be computed locally if all of its descendants are local or locally computable. A local subquery corresponds to any subgraph of the corresponding DAG such that all terminal nodes in the subgraph are locally accessible. If the DAG is a tree the subgraph will be a subtree. A remote subquery corresponds to any subgraph of the corresponding DAG such that no nodes are locally accessible.

Assume that queries 3.1 and 3.2 originate at Host A with relations CINEMAS and CINEFILM available locally. In the case of query 3.1, the subtree with root T1 is a remote subquery. CINEMAS and CINEFILM are local to Host A. Query 3.2 is entirely remote.

To obtain the local and remote decompositions we compute the transitive closure of RES and RNAME fields in QEXP. The transitive closure augments the DAG, adding new edges until each node has an edge to all of its descendants. Terminal nodes are viewed as descendants of themselves due to the *id* operation. The TRANS relation in figure 3.5 is the transitive closure of the QEXP relation of figure 3.4.

An expression is remote if no part of it is in common with the local schema. Remote relations and results can be identified as values of the RES field in TRANS such that no associated RNAME values are part of the local schema. A σ-join computing the null intersection of TRANS and SCHEMA will select the values of RES corresponding to remote RES values, namely those with no local descendants. The REMOTE relation in figure 3.5 is the null intersection of TRANS and the SCHEMA of Host A. The remote subquery is the subgraph of QEXP with remote values in the RES field.

An expression is local if all operands are present in the local schema. While calculating the local subexpression, temporary relations are not considered as operands as they are themselves derived from other non-temporary operands and do not appear in the schema. Local relations and results are computed using the TRANS$'$ relation. TRANS$'$ is the subgraph of TRANS with temporary values removed from the operand field, RNAME (see figure 3.5). In TRANS$'$, each node in RES is associated with all of the operand relations from which it is derived. If a result is only associated with local operand relations, it may be computed locally. The set subset selector σ-join of TRANS$'$ and SCHEMA computes the set of RES values such that the corresponding RNAME values are a subgraph of the schema. The LOCAL relation in figure 3.5 is the result of this join on the sample data. The tuples of QEXP with these values in the RES field form the locally executable subquery.

The following Aldat code decomposes QEXP into its local and remote components, LQEXP and RQEXP, using the local schema.

```
Procedure Decompose ,
Begin
    { Compute transitive closure of RES and RNAME in QEXP }
    TRANS ← (RES, RNAME in QEXP) ujoin
                        TRANS [RES icomp RNAME] (RES, RNAME in QEXP) ;
    { RQEXP will contain entirely remote subquery }
    REMOTE ← RES in TRANS ⊄ SCHEMA ;
    RQEXP ← QEXP ijoin REMOTE ;
    { Eliminate temporary relations in RNAME attribute of TRANS }
    TEMP ← RES where TYPE = 'T' in QEXP ;
    TRANS' ← TRANS [RNAME djoin RES] TEMP ;
    { LQEXP will contain entirely local subquery }
    LOCAL ← RES in TRANS' ⊂ SCHEMA ;
    LQEXP ← QEXP ijoin LOCAL ;
End Decompose ;
```

Figures 3.5 and 3.6 show the intermediate and final results of the above code on the SCHEMA and QEXP relations of figures 3.1 and 3.4.

TRANS (

RES	RNAME
LEE_FILMS	T3
LEE_FILMS	T2
LEE_FILMS	CINEMAS
LEE_FILMS	T1
LEE_FILMS	CINEFILM
LEE_FILMS	ACTORS
T3	T2
T3	CINEMAS
T3	T1
T3	CINEFILM
T3	ACTORS
T2	T1
T2	CINEFILM
T2	ACTORS
T1	ACTORS
ACTORS	ACTORS
CINEFILM	CINEFILM
CINEMAS	CINEMAS
FRED_ONLY	T4
FRED_ONLY	T5
FRED_ONLY	ACTORS
T4	ACTORS
T5	ACTORS

)

TRANS' (

RES	RNAME
LEE_FILMS	CINEMAS
LEE_FILMS	CINEFILM
LEE_FILMS	ACTORS
T3	CINEMAS
T3	CINEFILM
T3	ACTORS
T2	CINEFILM
T2	ACTORS
T1	ACTORS
ACTORS	ACTORS
CINEFILM	CINEFILM
CINEMAS	CINEMAS
FRED_ONLY	ACTORS
T4	ACTORS
T5	ACTORS

)

LOCAL (

RNAME
CINEFILM
CINEMAS

)

REMOTE (

RNAME
T1
ACTORS
FRED_ONLY
T4
T5

)

Fig. 3.5 Intermediate results of query decomposition.

RQEXP (	RES	OP	RNAME	ORD	TYPE)
	T1	where ACTOR='Lee Marvin'	ACTORS	U	T
	ACTORS	id	ACTORS	U	P
	FRED_ONLY	djoin	T4	L	R
	FRED_ONLY	djoin	T5	R	R
	T4	TITLE where ACTOR='Ginger Rogers'	ACTORS	U	T
	T5	TITLE where ACTOR='Fred Astaire'	ACTORS	U	T

LQEXP (	RES	OP	RNAME	ORD	TYPE)
	CINEFILM	id	CINEFILM	U	P
	CINEMAS	id	CINEMAS	U	P

Fig. 3.6 Final results of query decomposition.

The query is assumed to originate at Host A. Note that the union of RQEXP and LQEXP does not equal the original QEXP relation. This is because certain results are a combination of local and remote subexpressions.

3.3 The Query Dialogue

Once QEXP has been decomposed, the local subquery can be executed. The actual execution mechanism will not be discussed until the following chapter. For the present it is assumed that once an expression has been executed, all results are accessible at the host where the operation has been performed. A binary relation, RESULTS, lists the names of resulting relations in the RNAME attribute and the origin of the results in the SITE attribute. All RES values of the LQEXP appear in RESULTS with the value "A" in SITE as local results originate from Host A.

RESULTS also contains permanent relations resulting from the *rd* operation.

While the local subquery is being processed, the remote subquery is sent in parallel to the host's neighbour, in this case Host B. The neighbour repeats the decomposition and local execution phases of query processing on the incoming query relation. In our example Host B will be able to resolve the entire query sent by Host A in RQEXP. It then sends Host A its RESULTS relation which is appended to Host A's RESULTS relation as illustrated in figure 3.7.

RESULTS (RNAME	SITE)
CINEMAS	A
CINEFILM	A
T1	B
ACTORS	B
FRED_ONLY	B
T4	B
T5	B

Fig. 3.7 Local and remote results listed in RESULTS relation.

Host A may now ask Host B to send it the relations necessary to complete its knowledge. Not all results are necessary. For instance, Host A does not need the relations T4, T5 and ACTORS if it receives T1 and FRED_ONLY.

How can Host A determine what results it needs from its neighbour? Only those needed for remaining operations should be returned. At first glance it seems possible to decide at Host B what must be returned and to send it directly. It seems logical to return only the topmost results in the DAGs representing the remote subquery. In the given example this would work: T1 and FRED_ONLY would be returned as needed. However, suppose the original QEXP at Host A also contains the following expression:

"Find the films playing at the Roxy as well as the actors starring in them".

$$ROXY\_STARS \leftarrow TITLE \text{ where } CINEMA = \text{'Roxy' in } CINEFILM \text{ ijoin } ACTORS; \qquad 3.3$$

The corresponding expression DAG and augmented QEXP are shown in figures 3.8 and 3.9.

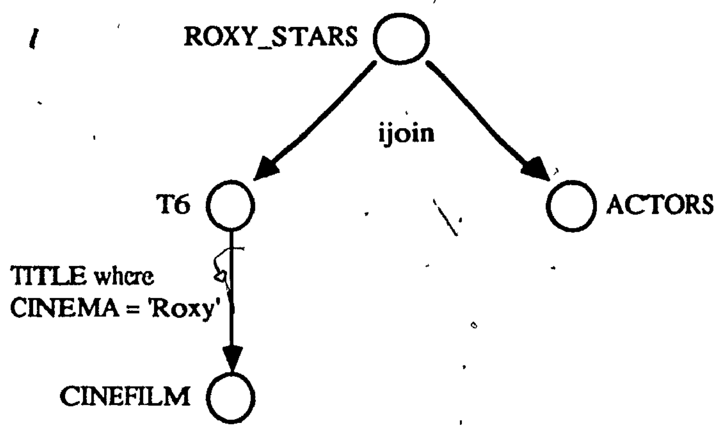


Fig. 3.8 Query expression 3.3.

The local and remote components of this expression are the relations T6 and ACTORS respectively. The decomposition of this new QEXP relation will produce the LQEXP relation shown in figure 3.10.

As ACTORS already figures in RQEXP, the addition of expression 3.3 to QEXP does not alter RQEXP. However, ACTORS must now be returned to Host A in order to compute the relation ROXY_STARS. It is the "left over" part of QEXP at Host A, combining local and remote subexpressions that determines what must be received from Host B to complete the original query. This residual query will be computed in the relation MQEXP ("M" for merge).

QEXP (RES	OP	RNAME	ORD	TYPE)
LEE_FILMS	[CINEMA, ADDRESS]	T3	U	R
T3	ijoin	T2	L	T
T3	ijoin	CINEMAS	R	T
T2	ijoin	T1	L	T
T2	ijoin	CINEFILM	R	T
T1	where ACTOR='Lee Marvin'	ACTORS	U	T
ACTORS	id	ACTORS	U	P
CINEFILM	id	CINEFILM	U	P
CINEMAS	id	CINEMAS	U	P
FRED_ONLY	djoin	T4	L	R
FRED_ONLY	djoin	T5	R	R
T4	TITLE where ACTOR='Fred Astaire'	ACTORS	U	T
T5	TITLE where ACTOR='Ginger Rogers'	ACTORS	U	T
ROXY_STARS	ijoin	T6	L	R
ROXY_STARS	ijoin	ACTORS	R	R
T6	Title where CINEMA = 'Roxy'	CINEFILM	U	T

Fig. 3.9 QEXP relation with the addition of expression 3.3.

LQEXP (RES	OP	RNAME	ORD	TYPE)
CINEFILM	id	CINEFILM	U	P
CINEMAS	id	CINEMAS	U	P
T6	TITLE where CINEMA = 'Roxy'	CINEFILM	U	T

Fig. 3.10 Local decomposition of QEXP relation in figure 3.9.

At this stage of query processing all relations in RESULTS are available, either locally or from Host B. Hence, the fourth possible value in the TYPE field of a query expression relation, "A" for available. Some of the relations listed in RESULTS may be final results such as FRED_ONLY in our example. Final results are identified and listed in a separate relation, FINAL_RESULTS, as they need no longer be taken into account. FINAL_RESULTS is of the same format as RESULTS.

To obtain MQEXP, all tuples with values in the RES field equal to relations in RESULTS are removed from QEXP. This eliminates edges from the DAG pertaining to operations which have already been executed. The relations in the RES field of the remaining tuples are those which have not yet been executed, as they are a combination of local and remote data. Redundant results may now be identified as those appearing neither as operands in the RNAME attribute of MQEXP, nor as final results. Redundant results are listed in the relation REDUNDANT_RESULTS. Now, RESULTS may be updated by eliminating redundancy. The RESULTS relation of the given example is shown before and after elimination of redundancy with FINAL_RESULTS in figure 3.11.

RESULTS (RNAME SITE)

RNAME	SITE
CINEMAS	A
CINEFILM	A
T6	A
T1	B
ACTORS	B
FRED_ONLY	B
T4	B
T5	B

(a)

FINAL_RESULTS (RNAME SITE)

RNAME	SITE
FRED_ONLY	B

REDUNDANT_RESULTS (RNAME SITE)

RNAME	SITE
T4	B
T5	B

RESULTS (RNAME SITE)

RNAME	SITE
CINEMAS	A
CINEFILM	A
T6	A
T1	B
ACTORS	B

(b)

Fig. 3.11 RESULTS relation (a) originally and (b) after elimination of redundant relations with FINAL_RESULTS and REDUNDANT_RESULTS.

MQEXP must then have a tuple appended to it for each pertinent result relation. The relations in RESULTS may now be considered as terminal nodes in new expression DAGs. The new nodes will have *id* in the OP field and "A" in the TYPE field. Figure 3.12 shows the MQEXP resulting from the QEXP of figure 3.9, the RQEXP of figure 3.6 and the LQEXP of figure 3.10. Figure 3.13 is the DAG representation of MQEXP.

MQEXP (RES	OP	RNAME	ORD	TYPE)
LEE_FILMS	[CINEMA, ADDRESS]	T3	U	R
T3	ijoin	T2	L	T
T3	ijoin	CINEMAS	R	T
T2	ijoin	T1	L	T
T2	ijoin	CINEFILM	R	T
ROXY_STARS	ijoin	T6	L	R
ROXY_STARS	ijoin	ACTORS	R	R
CINEMAS	id	CINEMAS	A	A
CINEFILM	id	CINEFILM	A	A
T6	id	T6	A	A
T1	id	T1	A	A
ACTORS	id	ACTORS	A	A

Fig. 3.12 MQEXP is the portion of QEXP combining local and remote results.

Host A will ask Host B to send it any relations in RESULTS where SITE indicates Host B. The tuples indicating these relations are selected into REQUEST. A REQUEST relation has a single attribute, RNAME. When a request relation is sent, the receiving host understands that it must return the relations named in RNAME to the sending host.

Determine_Request creates FINAL_RESULTS by selecting tuples from RESULTS such that the RNAME value does not appear in the RNAME attribute of

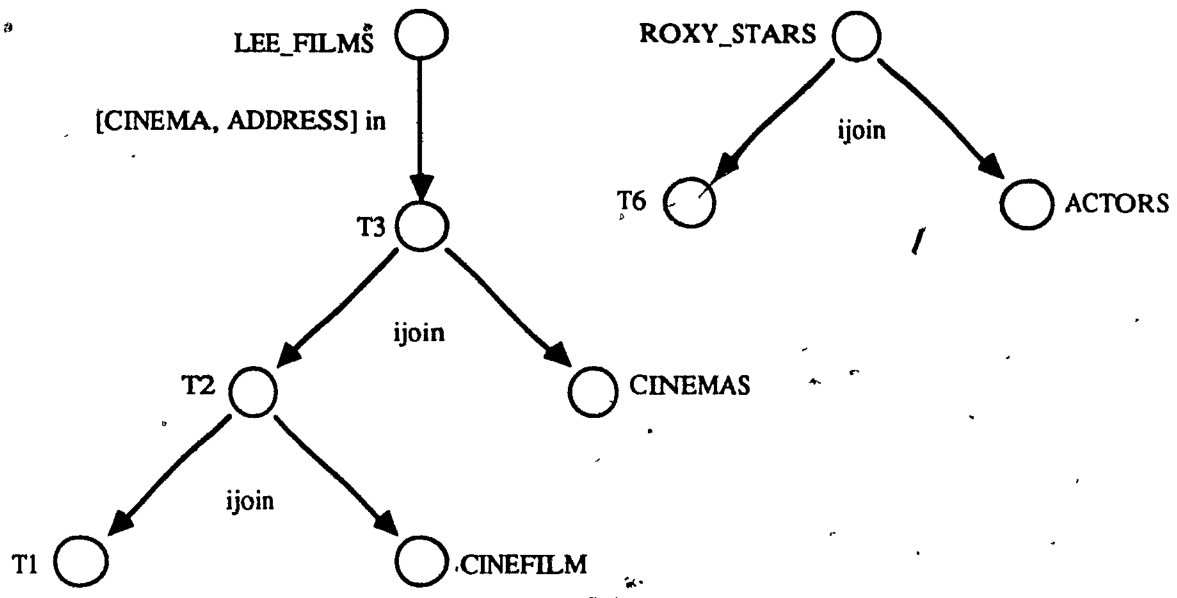


Fig. 3.13 DAG representation of MQEXP.

QEXP. In the DAG representation these values correspond to nodes with no predecessors. If the DAG is a forest, root nodes are selected. MQEXP is initialized by removing tuples identifying a relation in the RES attribute that has already been computed. Redundant results are then listed in REDUNDANT_RESULTS. Although redundant results are of no more concern in this chapter, they will be needed when query processing is extended to an arbitrary network. RESULTS is updated by removing redundancy and used to complete MQEXP.

```
Procedure Determine_Request ;
Begin
    FINAL_RESULTS ← RESULTS djoin QEXP
    { Eliminate operations that have already been executed }
    MQEXP ← QEXP [RES djoin RNAME] RESULTS ;
    { Select results not needed as operands }
    REDUNDANT_RESULTS ← (RESULTS djoin MQEXP) djoin FINAL_RESULTS ;
    { Eliminate redundant results from RESULTS }
    RESULTS ← RESULTS djoin REDUNDANT RESULTS ;
    { Complete MQEXP with id operation for terminal nodes }
    MQEXP [RES, OP, RNAME, ORD, TYPE ←+
                              RNAME, 'id', RNAME, 'U', 'A'] in RESULTS ;
End Determine_Request ;
```

Host A now sends a REQUEST relation to Host B. REQUEST lists remote relations appearing in the updated RESULTS relation, with any final results available at Host B. Two lines of Aldat code suffice to create REQUEST.

```
    REQUEST ← RNAME where SITE ≠ own in RESULTS ;
    REQUEST ←+ RNAME where SITE ≠ own in FINAL_RESULTS ;
```

Here and in upcoming routines, a constant, *own* refers to the name of the local host. In this case own has the value, "A". Host B will return all relations listed in REQUEST. Figure 3.14 shows the REQUEST relation derived from the given example.

REQUEST (RNAME)
T1
ACTORS
FRED_ONLY

Fig. 3.14 Resulting REQUEST relation to be sent to Host B.

Assuming the query was correct and referred to existing relations, Host A may now execute the remaining operations in MQEXP. To check that all necessary result relations are indeed present, a call to **Decompose** with MQEXP and RESULTS as parameters will produce a new LQEXP that should equal MQEXP. If not, the operations in MQEXP that do not appear in LQEXP are unresolvable as the data is present neither locally nor remotely.

Chapter 4 Executing Query Relations

Once we have expressed queries as relations we must find a way to execute them. QEXP relations offer a canonical format for representing arbitrary queries. If we also have a canonical representation of data relations, we may build a general execution program to execute QEXP relations on canonical data. Another advantage of canonical data is a constant format for transmitting and receiving data relations.

The execution program may be implemented by adding a frontend to an existing Aldat interpreter such as Relix [LAE 86]. The frontend reconstructs operand relations from their canonical format and reconstructs Aldat expressions from QEXP relations. Aldat may then interpret the expressions as usual. Figure 4.1 illustrates this logic. This chapter will also explore the construction of the canonical relation from existing data relations and the conversion of Aldat expressions to QEXP relations.

4.1 A Canonical Representation of Relations

A relational database may be represented in one relation of format:

CR (RNAME TID DNAME VAL)

Where RNAME is the name of the relation, TID identifies tuples, DNAME is the

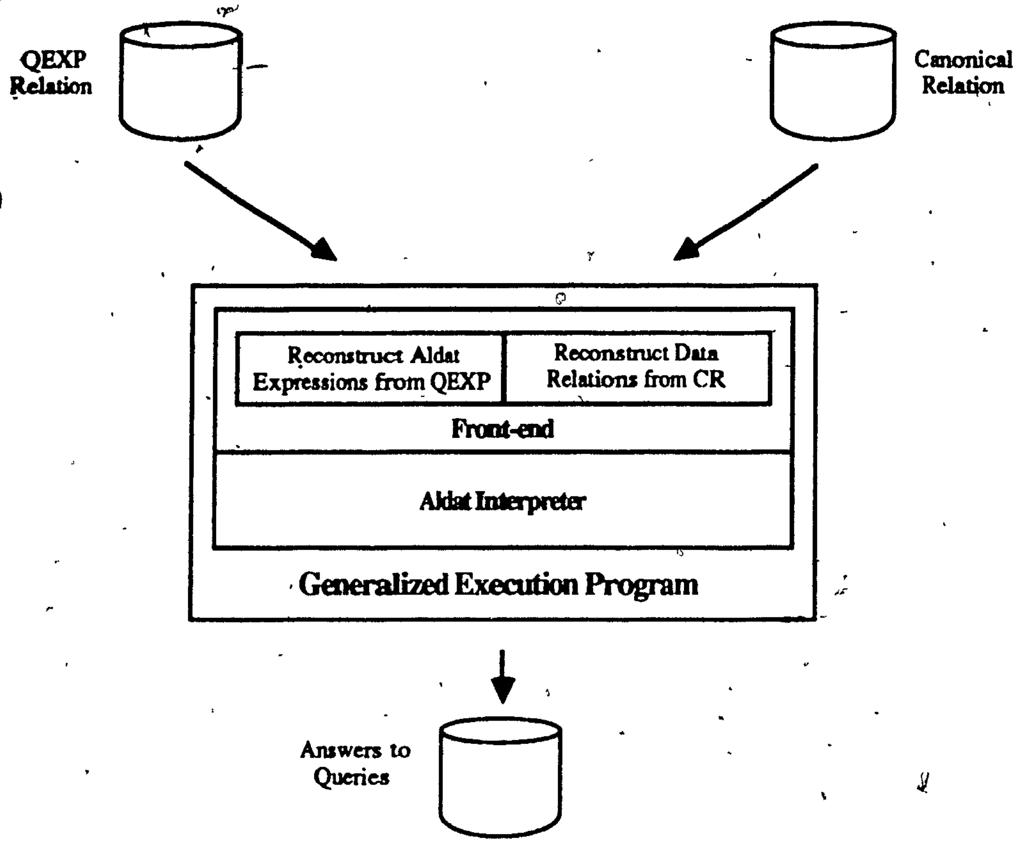

QEXP Relation
Canonical Relation
Reconstruct Aldat Expressions from QEXP
Reconstruct Data Relations from CR
Front-end
Aldat Interpreter
Generalized Execution Program
Answers to Queries

Fig. 4.1 Executing QEXP relations on canonical data.

name of an attribute and VAL is the attribute's value for the given tuple. Figure 4.2 shows the canonical representation, CR, of the CINEMAS and CINEFILM relations.

Note that the projection of CR onto RNAME and DNAME equals the SCHEMA as described in Chapter 3. The CR relation is actually data and schema in one. Although TID is an integer field, it is only for identification and imposes no ordering, as tuples in relations are unordered by definition.

4.2 Extensions to Aldat

The Relix implementation of Aldat imposes the relation as the unique unit of data [LAL 86]. We propose the addition of scalar variables such as integer and string.

CINEFILM (TITLE CINEMA SCREEN)

TITLE	CINEMA	SCREEN
Swing Time	Roxy	1
Rembetiko	Roxy	2
L'Avare	Odeon	1

CINEMAS (CINEMA ADDRESS SCREENS)

CINEMA	ADDRESS	SCREENS
Roxy	123 Main St.	2
Odeon	987 Park Ave.	1

CR (RNAME TID DNAME VAL)

RNAME	TID	DNAME	VAL
CINEMAS	1	CINEMA	Roxy
CINEMAS	1	ADDRESS	123 Main St
CINEMAS	1	SCREENS	2
CINEMAS	2	CINEMA	Odeon
CINEMAS	2	ADDRESS	987 Park Ave.
CINEMAS	2	SCREENS	1
CINEFILM	1	TITLE	Swing Time
CINEFILM	1	CINEMA	Roxy
CINEFILM	1	SCREEN	1
CINEFILM	2	TITLE	Rembetiko
CINEFILM	2	CINEMA	Roxy
CINEFILM	2	SCREEN	2
CINEFILM	3	TITLE	L'Avare
CINEFILM	3	CINEMA	Odeon
CINEFILM	3	SCREEN	1

Fig. 4.2 Relations CINEMAS and CINEFILM with their canonical representation.

In both the QEXP relations of Chapter 3 and the CR relation of section 4.1, names of relations and attributes appear as attribute values. To execute a QEXP relation these values must be retrieved and used as operands in Aldat expressions. We therefore propose to augment Aldat with two functions, **Pick** and **Scalar**, and with two more types of scalar variables, **rel_exp** and **dom_exp**.

The **Pick** function [MER 76] is logically equivalent to picking a tuple at random from a relation without replacing it. Thus, a tuple is only picked once. It provides a means of looping through a relation one tuple at a time. Random selection is appropriate as tuples are not ordered within a relation. For example, the expression:

ATUP ←**Pick** *(CINEMAS)* ;

assigns a unique tuple from CINEMAS to ATUP. As input, **Pick** accepts the name of a relation. It returns a singleton, which is a relation containing one tuple. The same tuple will not be returned twice. A **Reset** procedure is required to initialize the picking process on a given relation.

The **Scalar** function accepts a relational expression identifying one attribute of a singleton and assigns the value of the attribute to a scalar variable. For example, suppose the relation CINEMAS is as illustrated in figure 4.2. and the scalar variable, i, is declared as an integer. The statement:

i ← **Scalar** *(SCREENS in* **Pick** *(where CINEMA = 'Roxy' in CINEMAS));*

will assign the value "2" to variable i. The **Pick** function guarantees the parameter expression to be a singleton even if there is another Roxy cinema in the CINEMAS relation with a different number of screens. The type of the scalar variable must not conflict with the type of the attribute from which its value is being extracted.

Variables of type **rel_exp** may be assigned a character string corresponding to the name of an existing relation or to a valid relational expression. The following are examples of valid assignments to a **rel_exp** variable, r.

r ← *'TEMP';*
r ← *'CINEMAS ijoin CINEFILM';*

Variables of type **dom_exp** are assigned character strings containing the name of an attribute or a list of attributes separated by commas. Some valid assignments to a **dom_exp** variable, d, are:

d ← *'ADDRESS';*
d ← *'CINEMA, ADDRESS';*

When a **rel_exp** or **dom_exp** variable is encountered in an Aldat expression, it is replaced by its value. For example,

r ← '*CINEMAS*' ;
d ← '*CINEMA, ADDRESS*' ;
CINEADD ← *d in r* ;

is equivalent to

CINEADD ← *CINEMA, ADDRESS in CINEMAS* ;

These variables are used as formal parameters to metacode routines designed to execute on arbitrary relations. In Chapter 3 we mentioned that the **Decompose** procedure is to be executed once using the QEXP and SCHEMA relations, then again using the MQEXP and RESULTS relations. The procedure heading should therefore specify two parameters of type **rel_exp** as follows:

Procedure **Decompose** (*qexp, schema* : **rel exp**) ;

Note that the resulting relations, LQEXP and RQEXP are not parameters. We assume that relations are global and may be referenced by any procedure or function. **Decompose** is first called with parameters:

Decompose ('*QEXP*', '*SCHEMA*') ;

then with parameters:

Decompose ('*MQEXP*', '*RESULTS*') ;

4.3 Transforming Relations Between Representations

The extensions proposed in section 4.2 prove useful when converting relations to and from their canonical representations. Names of relations and attributes are extracted from the local schema while constructing CR, and from CR while rebuilding the original relation. By first calling the appropriate conversion routine, any data relation may be manipulated in either its original or canonical representation.

4.3.1 Converting Relations to Canonical Format

To convert a relation from its original form to canonical form, it is necessary to project onto each of its attributes in turn. A virtual attribute TID, containing a unique value for each tuple, is included in the projection to provide a means of associating values of different attributes to form tuples. The high level description of the algorithm is:

1. Uniquely identify each tuple with a virtual attribute TID.
2. For each attribute:
    - 2.1 Project onto TID and attribute.
    - 2.2 Append projection, relation name and attribute name to CR.

The procedure **Original Canonical** converts a relation to canonical format. It receives a character string identifying a relation in the local schema The result is appended onto the canonical relation CR, which initially may or may not be empty. **Original Canonical** is presented as a metacode procedure combining Aldat operations with the special functions and typed variables described in section 4.2.

```
Procedure Original_Canonical (relname : string) ;
Var
    r : rel_exp ; d : dom_exp ; i : integer ;
Begin
    r ← relname ;   { r will refer to parameter relation }
    { Select subset of schema pertaining to desired relation }
    SCHEMA' ← where RNAME = relname in SCHEMA ;
    { Make a copy of parameter relation adding a TID attribute }
    i ← 1 ;        let TID be i ;
    Reset (r) ;        ATUP ← Pick (r) ;
    { ATUP controls iteration for every tuple of parameter relation }
    While ATUP not empty
        { Append TID attribute to tuple and add to COPY }
        COPY ← ATUP ijoin (TID in ATUP) ;
        i ← i + 1 ;
        ATUP ← Pick (r) ;
    End While ;
    { ATUP controls iteration for every tuple of SCHEMA' relation }
    Reset (SCHEMA') ;
    ATUP ← Pick (SCHEMA') ;
    While ATUP not empty
        { Extract attribute name from ATUP }
        d ← Scalar (DNAME in ATUP) ;
        { Project COPY onto TID and current attribute }
        COLUMNS ← [d, TID] in COPY ;
        { Update canonical relation }
        CR [RNAME, TID, DNAME, VAL ← RNAME, TID, DNAME, d] in
                                        (ATUP ijoin COLUMNS) ;
        ATUP ← Pick (SCHEMA') ;
    End While ;
End Original Canonical ;
```

Figure 4.3 traces **Original Canonical**'s first *while* loop on the CINEMAS relation of figure 4.2. Tuples are picked one at a time from the parameter relation and copied with an additional attribute, TID, to the relation COPY. Note that the virtual attribute TID must be actualized before it can participate in the natural join. The **Pick** function extracts the tuples into ATUP. When ATUP is empty it signifies that there are no more tuples to be picked. TID takes its value from the scalar variable, i, which is incremented at each iteration. The natural join in this loop is actually the cartesian product as the operands have no common attributes. As both operands contain a single tuple, the cartesian product simply joins them together. In this way, a unique value of TID is appended to each tuple of the parameter relation.

7

Before entering first while loop:

relname $\boxed{\text{`CINEMAS'}}$

SCHEMA' (RNAME DNAME)

RNAME	DNAME
CINEMAS	CINEMA
CINEMAS	ADDRESS
CINEMAS	SCREENS

i $\boxed{1}$

ATUP (CINEMA ADDRESS SCREENS)

CINEMA	ADDRESS	SCREENS
Roxy	123 Main St.	2

After first iteration :

COPY (CINEMA ADDRESS SCREENS TID)

CINEMA	ADDRESS	SCREENS	TID
Roxy	123 Main St.	2	1

i $\boxed{2}$

ATUP (CINEMA ADDRESS SCREENS)

CINEMA	ADDRESS	SCREENS
Odeon	987 Park Ave.	1

After second and final iteration of first while loop :

COPY (CINEMA ADDRESS SCREENS TID)

CINEMA	ADDRESS	SCREENS	TID
Roxy	123 Main St.	2	1
Odeon	987 Park Ave.	1	2

i $\boxed{3}$

ATUP (CINEMA ADDRESS SCREENS)

CINEMA	ADDRESS	SCREENS
-	-	-

Fig. 4.3 Trace of first *while* loop of **Original_Canonical** ('CINEMAS').

Figure 4.4 continues the trace. **Original_Canonical**'s second *while* loop extracts the attribute names of the parameter relation from SCHEMA$'$, the subset of SCHEMA pertaining to the parameter relation. SCHEMA$'$ contains a tuple for each attribute of the parameter relation. We loop through SCHEMA$'$ with the **Pick** function as before. At each iteration, COPY is projected onto a different attribute from SCHEMA$'$ as well as TID. Once again a natural join is used to obtain a cartesian product, this time of the single tuple relation, ATUP, containing RNAME and

Before entering second while loop :

relname

$\boxed{\text{'CINEMAS'}}$

SCHEMA' (

RNAME	DNAME)
CINEMAS	CINEMA
CINEMAS	ADDRESS
CINEMAS	SCREENS

ATUP (

RNAME	DNAME)
CINEMAS	CINEMA

After first iteration :

d

$\boxed{\text{'CINEMA'}}$

COLUMNS (

CINEMA	TID)
Roxy	1
Odeon	2

CR (

RNAME	TID	DNAME	VAL)
CINEMAS	1	CINEMA	Roxy
CINEMAS	2	CINEMA	Odeon

ATUP (

RNAME	DNAME)
CINEMAS	ADDRESS

After second iteration :

d

$\boxed{\text{'ADDRESS'}}$

COLUMNS (

ADRESS	TID)
123 Main St.	1
987 Park Ave.	2

CR (

RNAME	TID	DNAME	VAL)
CINEMAS	1	CINEMA	Roxy
CINEMAS	2	CINEMA	Odeon
CINEMAS	1	ADDRESS	123 Main St.
CINEMAS	2	ADDRESS	987 Park Ave.

ATUP (

RNAME	DNAME)
CINEMAS	SCREENS

After third and final iteration of second while loop :

d

$\boxed{\text{'SCREENS'}}$

COLUMNS (

SCREENS	TID)
2	1
1	2

CR (

RNAME	TID	DNAME	VAL)
CINEMAS	1	CINEMA	Roxy
CINEMAS	2	CINEMA	Odeon
CINEMAS	1	ADDRESS	123 Main St.
CINEMAS	2	ADDRESS	987 Park Ave.
CINEMAS	1	SCREENS	2
CINEMAS	2	SCREENS	1

ATUP (

RNAME	DNAME)

Fig. 4.4 Trace of Original_Canonical ('CINEMAS') continued to second *while* loop.

DNAME attributes, and the COLUMNS relation resulting from the previous projection. This serves to append RNAME and DNAME to each tuple of columns. The result is ready to append to the canonical relation.

4.3.2 Restoring Relations to Their Original Format

Relations are converted from their canonical representations to their original format using the **Canonical Original** function. A call to **Canonical_Original** to restore CINEMAS to its original format looks like this:

$$CINEMAS \leftarrow \textbf{Canonical\_Original}\ (\text{'}CINEMAS\text{'})\ ;$$

The **Canonical Original** function accepts as a parameter, a character string containing the name of a single relation in CR and returns the relation in its original format. A call to **Canonical_Original** to convert the relation, CINEMAS, to its original format is traced in figure 4.5. The "||" operator signifies string concatenation.

```
Function Canonical_Original ( relname : string ) : relation ;
Var
    d, dd : dom_exp ;
Begin
    { Extract relation in canonical format from CR }
    CR' ← where RNAME = relname in CR ;
    { Separate schema from data and identify tuples}
    SCHEMA' ← RNAME, DNAME in CR' ;
    { Relation will be reconstructed in TEMP }
    TEMP ← TID in CR' ;
    { ATUP will hold a tuple for each attribute of relation }
    ATUP ← Pick (SCHEMA') ;
    While ATUP not empty
        { Extract name of attribute }
        d ← Scalar (DNAME in ATUP) ;
        { Append to list of attributes }
        if dd = ''
           then dd ← d
           else dd ← dd||','||d ;
        { Select tuples pertaining to current attribute }
        let d be VAL ;
        TEMP' ← TID, d where DNAME = Scalar (DNAME in ATUP) in CR' ;
        { Join attribute to reconstruction }
        TEMP ← TEMP ijoin TEMP' ;
        ATUP ← Pick (SCHEMA') ;
    End While ;
    { Update local schema, project out TID, return result }
    SCHEMA ← + SCHEMA' ;
    Canonical Original ← dd in TEMP ;
End Canonical Original ;
```

The canonical representation of the parameter relation is selected into CR$'$, then projected onto RNAME and DNAME to obtain the relation's schema. A relation, TEMP is initialized to contain the TID attribute. The *while* loop iterates once for each attribute of the relation being converted. The name of the current attribute is assigned to the **dom exp** variable, d. The current attribute name is appended to a list of attributes in another **dom exp** variable, dd. TEMP$'$ receives the tuples of CR$'$ relevant to the current attribute in ATUP. The relation's original representation is built up by joining TEMP$'$ with TEMP on their common attribute, TID. TEMP gets wider at each iteration. When the loop terminates, the local schema is updated with information for the reconstructed relation. TEMP is projected onto all of its attributes except TID and returned as the result.

Before entering loop:

CR' (RNAME TID DNAME VAL)

RNAME	TID	DNAME	VAL
CINEMAS	1	CINEMA	Roxy
CINEMAS	1	ADDRESS	123 Main St
CINEMAS	1	SCREENS	2
CINEMAS	2	CINEMA	Odeon
CINEMAS	2	ADDRESS	987 Park Ave.
CINEMAS	2	SCREENS	1

SCHEMA' (RNAME DNAME)

RNAME	DNAME
CINEMAS	CINEMA
CINEMAS	ADDRESS
CINEMAS	SCREENS

TEMP (TID)

TID
1
2

ATUP (RNAME DNAME)

RNAME	DNAME
CINEMAS	CINEMA

After first iteration :

attr	allattr	d
'CINEMA'	'CINEMA'	'CINEMA'

TEMP' (TID CINEMA)

TID	CINEMA
1	Roxy
2	Odeon

TEMP (TID CINEMA)

TID	CINEMA
1	Roxy
2	Odeon

ATUP (RNAME DNAME)

RNAME	DNAME
CINEMAS	ADDRESS

After second iteration :

attr	allattr	d
'ADDRESS'	'CINEMA, ADDRESS'	'ADDRESS'

TEMP' (TID ADDRESS)

TID	ADDRESS
1	123 Main St.
2	987 Park Ave.

TEMP (TID CINEMA ADDRESS)

TID	CINEMA	ADDRESS
1	Roxy	123 Main St.
2	Odeon	987 Park Ave.

ATUP (RNAME DNAME)

RNAME	DNAME
CINEMAS	SCREENS

After third iteration :

attr	allattr	d
'SCREENS'	'CINEMA, ADDRESS, SCREENS'	'SCREENS'

TEMP' (TID SCREENS)

TID	SCREENS
1	2
2	1

TEMP (TID CINEMA ADDRESS SCREENS)

TID	CINEMA	ADDRESS	SCREENS
1	Roxy	123 Main St.	2
2	Odeon	987 Park Ave.	1

ATUP (RNAME DNAME)

RNAME	DNAME
-	-

Final result returned to CINEMAS relation :

CINEMAS (CINEMA ADDRESS SCREENS)

CINEMA	ADDRESS	SCREENS
Roxy	123 Main St.	2
Odeon	987 Park Ave.	1

Fig. 4.5 Trace of CINEMAS ←Canonical_Original ('CINEMAS').

An entire CR relation may be converted to data relations in their original format by calling **Canonical_Original** for every RNAME value in CR. **Reconstruct Relations** loops through CR reconstructing all relations from their canonical format.

```
Procedure Reconstruct Relations ;
Begin
    REL_LIST ← RNAME in CR ;
    ATUP ← Pick (REL_LIST) ;
    While ATUP not empty
        Scalar (ATUP) ← Canonical_Original (Scalar (ATUP)) ;
        ATUP ← Pick (REL_LIST) ;
    End While ;
End Reconstruct Relations ;
```

4.4 Converting Aldat Expressions to and from their Relational Representations

In this section we call on data structures and semantic analysis techniques described in [AHO 77]. In the Relix Aldat implementation of Aldat [LAL 86], an unambiguous LALR-1 grammar is proposed and used to perform semantic analysis of Aldat expressions and generate intermediate code executable by an interpreter. The same grammar and semantic analysis techniques could be used to generate a QEXP relation from an Aldat expression. The graphic representations of Aldat expressions in Chapter 3 are very similar to parse trees. In fact, if common subexpressions are considered as identical but distinct nodes, and the "id" operator is omitted, expressions are always trees.

4.4.1 Converting Aldat Expressions to QEXP-Relations

We wish to represent strings of the form:

new relation ← Aldat expression ;

as QEXP relations, where new_relation is to hold the result of the Aldat expression on the right side of the assignment operator. The Aldat expression is parsed. When the parser encounters a string of characters defining a relational operation without nested expressions, it adds the operation to a QEXP relation. The parser calls four basic procedures, **Temp Name**, **Add_Result**, **Replace** and **Complete**.

The **Temp_Name** function returns a system generated character string that is used to identify intermediate results. In the examples of Chapter 3, these results had names starting with "T" for "Temp" such as "T1"and "T2". No two intermediate results contained in the same QEXP relation may have the same name.

Add_Result takes two parameters: the first is the name of a result relation, either intermediate or final; the second is the expression producing the result. The expression is divided into operator and operands and added to a QEXP relation.

The **Replace** procedure replaces its first parameter, which is a nested expression, with its second parameter, the name identifying the expression generated by **Temp_Name**. The third parameter is the expression in which the replacement is to occur.

Complete is called to add the "id" operations as described in section 3.1. A high level description of the conversion algorithm follows. The Aldat expression being scanned is in the character string, Aldat_exp. It is being assigned to a relation named new_rel.

```
1. While there are nested expressions:
    1.1 in a left to right scan find the first subexpression without nested ex-
        pressions as arguments, assign it to sub_exp.
        1.2 temp ← Temp Name
        1.3 Add_Result (temp, sub_exp)
        1.4 Replace (sub_exp, temp, Aldat exp)
    2. Add Result (new rel, Aldat exp)
    3. Complete
```

4.4.2 Rebuilding an Aldat Expression from a QEXP Relation

The QEXP relation contains fragments of Aldat expressions in the RNAME, OP and RES fields. The ORD and TYPE fields indicate how fragments are related to each other. The original Aldat expression may be reconstructed by pasting fragments together with the addition of parentheses and the keyword "in".

The recursive function **Build Exp** returns an Aldat expression reconstructed from its relational representation. The **Scalar** function is used to extract fragments from the QEXP relation. The **Card** function accepts the name of a relation and returns its cardinality as a scalar value.

```
Function Build_Exp (relname : string) : string ;
Var
    op, rr, rl : string ;
Begin
    { Select tuples defining operation giving parameter relation }
    TEMP ← where RES = relname in QEXP ;
    { Extract operator }
    op ← Scalar (OP in TEMP) ;
    { If operator is "id" return name of relation }
    If op = 'id
      then Build_Exp ← relname
      else
    { Recursive call for unary operations }
        if Card (TEMP) = 1
          then rr ← Scalar (RNAME in TEMP)
                Build_Exp ← '(' || op || 'in' || Build_Exp (rr) || ')'
          { Recursive call for binary operations }
          else rr ← Scalar (RNAME where ORD = 'R' in TEMP) ;
              rl ← Scalar (RNAME where ORD = 'L' in TEMP) ;
              Build_Exp ← '(' || Build_Exp (rl) || op || Build_Exp (rr) || ')' ;
End Build_Exp ;
```

The recursion stops when the *id* operation is encountered. The name of the relation is returned to the caller. All paths in QEXP end with an identity operator, so the recursion will stop in all cases. When the parameter relation is produced by a unary operation, the original expression may be reconstructed from QEXP by concatenating the string present in the OP field, followed by "in", followed by the string identifying the operand expression. As operands may themselves be expressions, they are computed recursively. Similarly, an expression defining a binary operation may be built up from QEXP by concatenating the left operand with the OP field and the right operand. Expressions are enclosed in parentheses to avoid any ambiguity with respect to the order of evaluation.

Chapter 5 Distributed Query Processing in an Arbitrary Network

In Chapter 3 the database was restricted to two hosts. When a host lacked sufficient data to answer a query, its only option was to send the remote component to its neighbour, then wait and see if the neighbour could complete the request. In an arbitrary network, a host may have several neighbours. As there is no global schema, a host has no way of knowing which neighbour, if any, has the data needed to complete its knowledge.

5.1 Query Propagation

Consider the network of figure 5.1. Each node represents a host that has access to a subset of the database. All edges represent two-way channels of communication between hosts. When a host is unable to resolve a query it may send the remote decomposition to all of its neighbours. The neighbours may in turn send unresolved subqueries to their neighbours. Successive decompositions propagate from host to host until the query is entirely resolved or until all hosts have been approached. Imagine the following scenario.

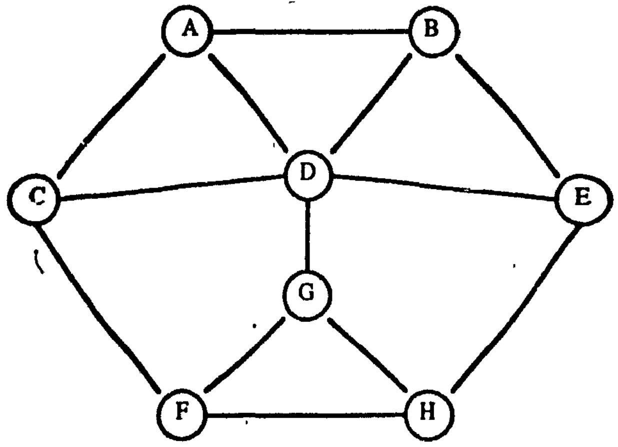


Fig. 5.1 A sample network.

A query originates at Host D. Unable to respond entirely, D forwards the remote decomposition to hosts A, B, C, G and E. Neither, C nor G can completely resolve the remote decomposition nor do they know if it has been resolved at another location. It is therefore reasonable for them to send remote decompositions to their neighbours. However, if both C and G query all neighbours, F and A will be queried redundantly. Moreover, it would not make sense for either of them to send a request back to Host D.

To ensure all nodes are approached at most once, we impose a spanning tree on the network as in figure 5.2. A host may pass remote subqueries to all of its neighbours in the tree, except the one from which it received the request. The sending node is regarded as the parent of the receiving nodes in the tree's hierarchy. Each node has a single parent and therefore will only receive one subquery. As there are no cycles in a tree, the query propagation will stop at the leaves.

Note that the tree structure is entirely independent of the physical layout of the

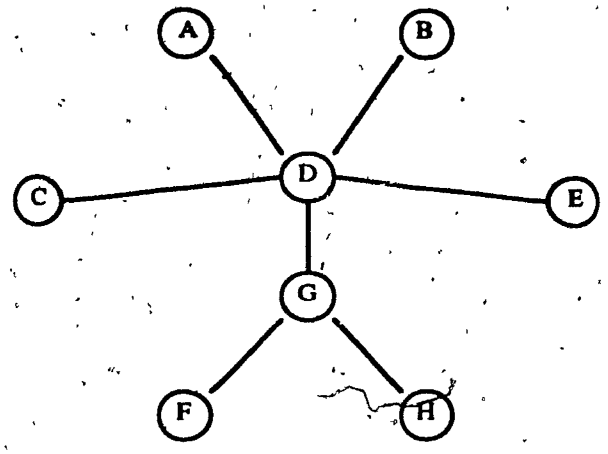


Fig. 5.2 A spanning tree of the sample network.

network. A "neighbour" is simply the address of a host that may be approached to complete a query. The tree must be defined before processing a query, in as much as each host must know who its neighbours are. The tree structure need not be respected for communications unrelated to query resolution.

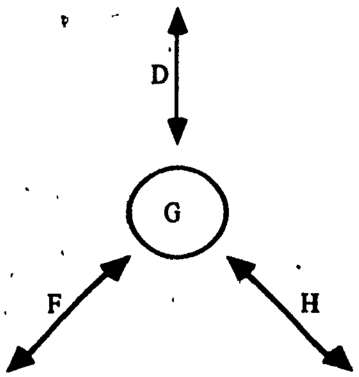
5.2 Local Participation in Query Processing

Query processing will now be explored from the point of view of an individual host and its responses to different events in the network. If Aldat is augmented with a mechanism that can react to network events, the relational framework can also deal with the distribution of query processing.

5.2.1 Special Purpose Functions and Relations

We propose to have Aldat communicate with the network via a system relation, FROM, and procedures **Receive** and **Send**. An **Execute** procedure activates the general execution program discussed in Chapter 4. The host knows its position in the network through a unary relation, NEIGHBOURS, containing in its attribute, SITE, the names of all nodes with which a host may communicate. The neighbours relation at Host G is shown in figure 5.3. A list of neighbours to which a remote decomposition may be sent is obtained by removing the sending neighbour from NEIGHBOURS. The sending neighbour is the one appearing in the SITE attribute of FROM, coupled with a QEXP relation.

When a host receives a relation from one of its neighbours, a tuple appears in FROM. Its two attributes, SITE and RTYPE, refer to the name of the neighbour sending the relation and the type of relation being sent. RTYPE can contain four values: QEXP for a query relation, CR for a canonical relation, RESULTS for a result relation, and REQUEST for a list of relations to be returned to the sending host. Each type of relation prompts a different action. The FROM relation at Host G is represented in figure 5.3. It is shown signaling the arrival of a QEXP relation arriving from Host D.



```
SCHEMA ( RNAME  DNAME )
            R2       ...

NEIGHBOURS ( SITE )
               D
               F
               H

FROM ( SITE  RTYPE )
         D    QEXP
```

Fig. 5.3 Metadata relations at Host G.

FROM has some particular features: it never contains more than one tuple; and the function **Receive** erases it. **Receive** integrates an incoming relation, described in FROM, with the local database. Before an incoming relation may be accessed it must be received. For example, when a tuple appears in FROM indicating an incoming QEXP relation, the statement:

QEXP ← **Receive** ;

tells Aldat to receive the incoming relation and to name it "QEXP".

The **Send** procedure sends a single relation to one or more neighbours. Its three parameters are character strings. The first parameter is the name of the relation being sent. The second names a subset of the NEIGHBOURS relation listing the destination hosts. The third parameter specifies the type of the relation being sent. The types correspond to those allowed in the RTYPE attribute of FROM. The FROM relation, the **Send** procedure and the **Receive** function form Aldat's network interface.

Execute receives the name of a QEXP relation, reconstructs the corresponding Aldat expressions and executes them. Parameter relations must be locally available in original or canonical format. Those in canonical format will be reconstructed as described in Chapter 4.

5.2.2 Local Response to an Incoming QEXP Relation

A single procedure will control query processing from the moment a QEXP relation arrives in FROM until the list of results is returned to the sender. When FROM signals the arrival of a QEXP relation, **Process_Query** is activated to deal with the request.

```
Event : QEXP relation is signaled in FROM
Procedure Process Query ;
Var local : boolean ;
Begin
    local ← Receive Query ;
    If not local then
        Receive_Results ;
        Determine_Request ;
        Request_Relations ,
        Receive Relations ;
        Merge ;
    Return_Results ;
End Process Query ;
```

The functions and procedures used in **Process_Query** are illustrated in a scenario taking place at Host G. The local configuration is as shown in figure 5.3. Host G is unaware that relation R3 is located at Host F and relations R1 and R4 are located at Host H.

One data relation, R2 is locally accessible. The neighbours relation shows that G communicates with hosts D, F and H. The arrival of the QEXP relation signaled in FROM triggers the call to **Process Query**. Figure 5.4 shows an example of the incoming QEXP relation.

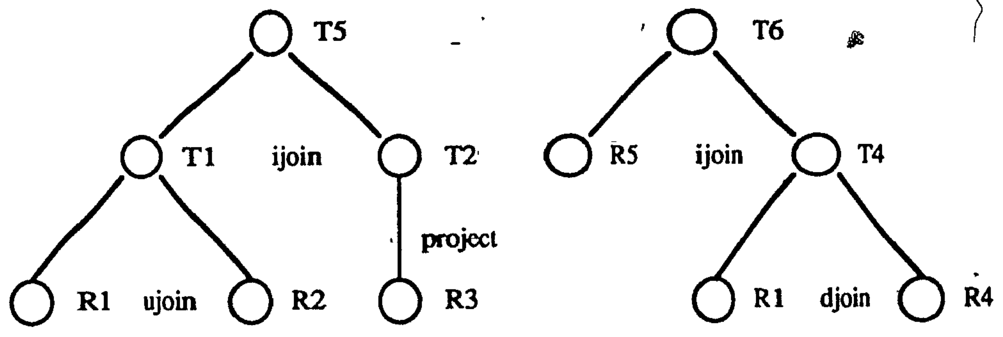
For the sake of simplicity we restrict network activity to the processing of one original QEXP relation. This ensures that all messages pertain to the same application.

5.2.2.1 Receive_Query

Process_Query initiates the transaction with a call to **Receive_Query**.

QEXP (RES	OP	RNAME	ORD	TYPE)
T5	ijoin	T1	L	T
T5	ijoin	T2	R	T
T1	ujoin	R1	L	T
T1	ujoin	R2	R	T
T2	X, Y in	R3	U	T
R1	id	R1	U	P
R2	id	R2	U	P
R3	id	R3	U	P
T6	ijoin	R5	L	T
T6	ijoin	T4	R	T
R5	id	R5	U	P
T4	djoin	R1	L	T
T4	djoin	R4	R	T
R4	id	R4	U	P

(a)



(b)

Fig. 5.4 (a) The incoming QEXP relation. (b) Its graph representation.

```
Function Receive_Query : boolean ;
Begin
    SENDER ← SITE in FROM ;
    QEXP ← Receive ;
    Decompose ('QEXP', 'SCHEMA') ;
    Execute ('LQEXP') ;
    RESULTS [RNAME, SITE ← RES, own] in LQEXP ;
    if RQEXP is empty
      then Receive_Query ← true
      else RECEIVERS ← NEIGHBOURS djoin SENDER ;
          Send ('RQEXP', 'RECEIVERS', 'QEXP') ;
          Receive_Query ← false ;
```

End **Receive_Query** ;

Figure 5.5 shows the results of **Receive_Query** on our example QEXP relation. A SENDER relation is established listing the name of the host sending the QEXP relation. The sender may be a neighbouring node or the local host itself. The QEXP relation is received in a relation called QEXP, then decomposed into local and remote components. The local subexpression is executed and the names of results are stored in a RESULTS relation. If the remote decomposition were empty, **Receive_Query** would return the boolean value, "true" signifying that the expression was entirely local. As the remote decomposition is not empty, a RECEIVERS relation is assigned the names of the hosts who are to receive RQEXP. RQEXP is then sent to all sites listed in RECEIVERS and **Receive_Query** returns "false".

When **Receive_Query** terminates, the boolean variable, *local*, is tested to determine if results are expected from neighbours. In this example *local* equals false so control is passed to **Receive_Results**.

LQEXP (RES	OP	RNAME	ORD	TYPE)
R2	id	R2	U	P

SENDER (SITE)
D

RQEXP (RES	OP	RNAME	ORD	TYPE)
T2	X, Y in	R3	U	T
R1	id	R1	U	P
R3	id	R3	U	P
T6	ijoin	R5	L	T
T6	ijoin	T4	R	T
R5	id	R5	U	P
T4	djoin	R1	L	T
T4	djoin	R4	R	T
R4	id	R4	U	P

RESULTS (RNAME	SITE)
R2	own

RECEIVERS (SITE)
F
H

local

$\boxed{\text{false}}$

Fig. 5.5 New relations created by **Receive_Query**.

5.2.2.2 Receive Results

Once the remote decomposition has been broadcast, the local host must wait for each receiver to respond by returning a RESULTS relation indicating what results it can contribute to query execution. If a receiver can contribute nothing to the query processing, it returns an empty RESULTS relation.

The **Receive_Results** procedure assembles the list of available results in RESULTS as responses are received from the different neighbours.

```
Procedure Receive_Results ;
Begin
    While RECEIVERS not empty
        Event : receiver responds
        { Delete responding host from RECEIVERS and receive response }
        RECEIVERS ← RECEIVERS djoin FROM ;
        RESULTS ←+ Receive ,
    End While ;
End Receive Results ;
```

Figure 5.6 traces **Receive Results** on the current example. Initially the RECEIVERS relation is as shown in figure 5.5. As it is not empty the process waits for a response. A response is signaled by a tuple appearing in FROM indicating a RESULTS relation coming in from Host F. (see figure 5.6.a) The answering host is deleted from RECEIVERS, leaving the names of neighbours from which answers have not yet been received. The relation is received and appended to the existing RESULTS relation. Figure 5.6.b shows the modified RECEIVERS and RESULTS relations.

RESULTS is still not empty so the process, once again, waits for a receiver to answer. This time FROM signals a RESULTS relation arriving from Host H. (see

FROM (SITE	RTYPE)
F	RESULTS

(a)

RECEIVERS (SITE)
H

RESULTS (RNAME	SITE)
R2	own
R3	F
T2	F

(b)

FROM (SITE	RTYPE)
H	RESULTS

(c)

RECEIVERS (SITE)
-

RESULTS (RNAME	SITE)
R2	own
R3	F
T2	F
R1	H
R4	H
T4	H

(d)

Fig. 5.6 Trace of **Receive_Results**. (a) Host F answers. (b) RECEIVERS and RESULTS after first iteration. (c) Host H answers. (d) RECEIVERS and RESULTS after second iteration.

figure 5.6.c) Host H is deleted from RECEIVERS and the incoming relation is received and appended to RESULTS as before. Figure 5.6.d shows relations RECEIVERS and RESULTS after the second iteration. The empty RECEIVERS relation indicates that all receivers have responded. **Receive_Results** is exited and **Process_Query** calls **Determine_Request.**

5.2.2.3 Determine_Request

The **Determine_Request** procedure has already been discussed in section 3.3.

It determines what results the local host needs to request from its neighbours. A call to **Determine_Request** sets aside final results in FINAL RESULTS, places the unresolved portion of the query in MQEXP, and removes redundancy from RESULTS. Figure 5.7 shows the results of **Determine_Request** on the sample data.

FINAL_RESULTS

RNAME	SITE
-	-

REDUNDANT_RESULTS

RNAME	SITE
R3	F
R4	H

RESULTS

RNAME	SITE
R2	own
T2	F
R1	H
T4	H

MQEXP

RES	OP	RNAME	ORD	TYPE
T5	ijoin	T1	L	T
T5	ijoin	T2	R	T
T1	ujoin	R1	L	T
T1	ujoin	R2	R	T
T6	ijoin	R5	L	T
T6	ijoin	T4	R	T
R5	id	R5	U	P
T4	id	T4	U	A
R1	id	R1	U	A
R2	id	R2	U	A
T2	id	T2	U	A

Fig. 5.7 Relations created by **Determine_Request** with the updated version of RESULTS

5.2.2.4 Request_Relations

The **Request_Relations** procedure requests relations needed to continue query processing from neighbouring nodes. These relations appear in the RNAME attribute of the updated RESULTS relation.

```
Procedure Request_Relations ;
Begin
    SITES ←SITE where SITE ≠ own in RESULTS ;
    Reset (SITES) ;
    TOLIST ←Pick (SITES) ;
    While TOLIST not empty
        REQUEST ←RNAME in TOLIST ijoin RESULTS ;
        SEND ('REQUEST', 'TOLIST', 'REQUEST') ;
        TOLIST ← Pick (SITES) ;
    End While ;
End Request_Relations,
```

Neighbours with needed results are selected into SITES. SITES is reset and a tuple identifying one neighbour is picked into TOLIST. At each iteration, REQUEST receives the names of relations needed from the current host in TOLIST. REQUEST is then sent to the host in TOLIST and another tuple is picked from SITES. Iteration stops when TOLIST is empty, indicating that all sites have been picked and have been sent a request.

Figure 5.8 traces **Request_Relations** on the sample problem.

SITES (SITE)
F
H

(a)

TOLIST (SITE)
F

(b)

REQUEST (RNAME)	TOLIST (SITE)
T2	H

(c)

REQUEST (RNAME)	TOLIST (SITE)
R1	
T4	

(d)

Fig. 5.8 Execution of **Request_Relations** on sample problem.
(a) The SITES relation. (b) Result of initial pick.
(c) REQUEST and TOLIST after first iteration. The REQUEST relation has been sent to Host F.
(d) REQUEST and TOLIST after second iteration. The REQUEST relation has been sent to Host H.

5.2.2.5 Receive_Relations

Once requests have been sent, the **Receive_Relations** procedure waits for the neighbours to respond by returning the requested relations in canonical format.

```
Procedure Receive_Relations ,
Begin
    While SITES not empty
        Event : CR arrives in FROM
        SITES ← SITES djoin FROM ;
        CR ←+ Receive ;
    End While ;
End Receive_Relations ;
```

Arriving data is appended to a canonical relation, CR which is initially empty. Each time a neighbour answers, its name is deleted from the the SITES relation that was created in **Request_Relations**. When SITES becomes empty, **Receive_Relations** knows that all requested data has been received and terminates.

Figure 5.9 illustrates the execution of **Receive_Relations** in the current scenario.

(a)

FROM (SITE	RTYPE)
H	CR

(b)

SITES (SITE)
F

CR (RNAME	...)
R1	...
⋮	
T4	...
⋮	

(c)

FROM (SITE	RTYPE)
F	CR

(d)

SITES (SITE)
-
-

CR (RNAME	...)
R1	...
⋮	
T4	...
⋮	
T2	...
⋮	

Fig. 5.9 Execution of **Receive_Relations** on sample problem. (a) Host H responds with requested data. (b) SITES and CR after first iteration. (c) Host H responds. (d) SITES and CR after second iteration.

5.2.2.6 Merge

Now that more data is locally available, **Process_Query** calls **Merge** to recompute a local decomposition using newly available relations. Available relations are those appearing in RESULTS. The unresolved portion of QEXP was placed in MQEXP when **Determine_Request** was called. **Merge** calls to **Decompose** with MQEXP and RESULTS as parameters to obtain a new LQEXP relation to execute locally. RESULTS is reset to contain the results of this latest execution.

```
Procedure Merge ,
Begin
    Decompose ('MQEXP', 'RESULTS') ;
    Execute ('LQEXP') ;
    RESULTS [RNAME, SITE ←RES, own] in LQEXP ;
End Merge ;
```

Figure 5.10 shows LQEXP and RESULTS relations after **Merge** is executed on the sample problem.

LQEXP (RES	OP	RNAME	ORD	TYPE)	RESULTS (RNAME	SITE)
T5	ijoin	T1	L	T	T5	own
T5	ijoin	T2	R	T	T1	own
T1	ujoin	R1	L	T	T4	own
T1	ujoin	R2	R	T	R1	own
T4	id	T4	U	A	R2	own
R1	id	R1	U	A	T2	own
R2	id	R2	U	A		
T2	id	T2	U	A		

Fig. 5.10 LQEXP and RESULTS produced by **Merge**.

5.2.2.7 Return_Results

The local host informs the sender that instigated the query processing of available results. FINAL_RESULTS, originally listing results of completed expressions, is augmented with the results of the last local execution. If the query processing originated locally, FINAL_RESULTS will be returned to the user. In section 3.3 we explained that the local host has no way of knowing what results are required by the sender. If the query processing did not originate locally it is possible that certain results considered as redundant locally may be needed by the sender. In this case REDUNDANT_RESULTS is also appended to FINAL_RESULTS. **Return_Results** returns the augmented FINAL_RESULTS to the host identified in SENDER. SENDER is the same relation created in **Receive_Query**.

```
Procedure Return_Results ;
Var name : string ;
Begin
    FINAL_RESULTS ←+ RESULTS ;
    name ←Scalar(SITE in SENDER) ;
    If name ≠ own then
        FINAL_RESULTS ←+ REDUNDANT_RESULTS ;
    Send ( 'FINAL_RESULTS', 'SENDER', 'RESULTS') ;
End Return_Results ;
```

Figure 5.11 shows the final results to be returned to Host D in our scenario.

FINAL_RESULTS (RNAME SITE)

RNAME	SITE
T5	own
T1	own
T4	own
R1	own
R2	own
T2	own
R3	F
R4	H

Fig. 5.11 FINAL_RESULTS relation produced by **Return_Results.**

5.2.3 Local Response to an Incoming REQUEST Relation

A host must also deal with a neighbour's request for data. A request takes the form of a REQUEST type relation being signaled in FROM. When a local host receives a REQUEST relation, it must assemble the requested data in canonical format and send it to the requesting node. **Process_Request** converts requested relations one at a time to canonical format, appending then to a CR relation which is initially empty. CR is then sent to the host requesting the data.

```
Event : REQUEST relation is signaled in FROM
Procedure Process_Request ;
Var relname : string ;
Begin
    TOLIST ← SITE in FROM ;
    REQUEST ← Receive ;
    Reset (REQUEST) ;
    ATUP ← Pick (REQUEST) ;
    CR ← empty ;
    While ATUP not empty
            relname ← Scalar (RNAME in ATUP) ;
            Original_Canonical (relname) ;
            ATUP ← Pick (REQUEST) ;
    End While ;
    Send ('CR', 'TOLIST', 'CR') ;
End Process_Request ;
```

Figure 5.12 shows how Host G deals with a REQUEST relation arriving from Host D.

5.3 Resolving Distributed Queries

The query processing strategy described in this chapter will resolve a distributed query providing that the query is correctly formulated and the network is one hundred

```
                FROM ( SITE   RTYPE )
                        D     REQUEST
                        (a)
        TOLIST ( SITE )        REQUEST (RNAME )
                D                        T4
                                         T5
        ATUP (RNAME )          CR ( RNAME ... )
                T4
                        (b)

        CR ( RNAME ... )       ATUP (RNAME )
             T4 ...                      T5
             :
                        (c)

        CR ( RNAME ... )       ATUP (RNAME )
             T4 ...                      :
             :
             T5 ...
             :
                        (d)
```

Fig. 5.12 Trace of **Process_Request**. (a) A REQUEST relation arrives from Host D. (b) Before entering loop. (c) After first iteration. (d) After second iteration.

percent reliable. Any syntactic errors in the query will be detected by the parser when the query is converted to a QEXP relation. Logical errors, such as references to non-existent relations, can not be detected until the query propagation has terminated without having resolved the initial query. If a host fails to respond in the query dialogue the process will wait indefinitely. A time-out mechanism is required to tell the host to stop waiting for a response after a reasonable delay.

If queries are to execute concurrently, each must be associated with its own **Process_Query** process. Messages must indicate to which process they pertain.

Chapter 6 Conclusion

A dialogue model of distributed query processing allows independent processors to resolve distributed queries with no prior knowledge of a global schema. Query processing is highly parallel in the dialogue model: subqueries may start executing at several hosts before all of the data is actually located.

We used this distributed query processing strategy as a vehicle to explore the applicability of Aldat. It appears that Aldat requires some extensions to be able to deal with the data distribution. We do not propose to implement this system in Aldat; a lower level language such as C would be much more appropriate. However, since Aldat is such a powerful formalism, it allowed us to illustrate the strategy in relatively few lines of code. Moreover, this rather primitive attempt at dealing with data distribution using the relational model gave us some valuable insight into the representation and manipulation of metadata and the development of metacode.

6.1 Applications to Metadata and Metacode

QEXP relations allow queries to be expressed and manipulated using the same formalism with which we represent and manipulate data. Thus, queries may be stored as data in the database. A canonical representation of queries motivated the study of a canonical data representation. The canonical relation combines data and metadata in a single format to represent arbitrary relations. If relation and attribute names may appear as data values, semantic information can be represented as metadata. Two examples of semantic information are integrity constraints and hierarchical relations among attributes. The QEXP relation effectively eliminates the distinction between data and code.

Any integrity constraint can be represented as a query in a sufficiently powerful query formalism [MER 84]. For instance, in the sample database of Chapter 2, all films appearing in the TITLE attribute of the ACTORS relation must have a corresponding tuple in the FILMS relation. Tuples of ACTORS violating this constraint can be detected by the Aldat query:

VIOLATIONS ← ACTORS djoin FILMS ;

which in turn may be represented as the QEXP relation of figure 6.1.

QEXP (	RES	OP	RNAME	ORD	TYPE)
	VIOLATIONS	djoin	ACTORS	L	R
	VIOLATIONS	djoin	FILMS	R	R
	ACTORS	id	ACTORS	U	P
	FILMS	id	FILMS	U	P

Fig. 6.1 An integrity constraint represented by a QEXP relation.

A hierarchical relationship may be defined on the attributes ACTOR and DIRECTOR of the relations ACTORS and FILMS of figure 2.1. Both of these attributes identify people. It is sometimes necessary to distinguish between people who are actors and people who are directors. In other situations the person's function may be irrelevant and the user will refer to people in general. Relations can represent such hierarchies In fact the expression DAGs represented in QEXP relations are also hierarchies. The relation in figure 6.2 shows that an actor is a person and that a director is a person. In [SMI 77] such abstractions are referred to as *generalizations.* The paper also discusses another useful data abstraction, *aggregation.*

GENERALIZATIONS (SUBJECT	ISA)
ACTOR	PERSON
DIRECTOR	PERSON

Fig. 6.2 PERSON is a generalization of ACTOR and DIRECTOR.

Once data and code have a common representation, metacode can be written to manipulate code. The **Decompose** procedure of Chapter 3 does precisely this. One can go a step further and write self-modifying code. Such capabilities, in the LISP language for example, have proven to be an indispensable tool in many artificial intelligence applications.

6.2 Future Work

It remains to study how much optimization can be introduced into the query dialogue. In the DBMSs discussed in Chapter 1, query processing is globally optimized at the query's site of origin. In some systems such as R*, local hosts can further optimize access to their own data. Most global query optimization strategies assume that the cost of transmission dominates the cost ~~of local~~ processing and access to secondary storage In any geographically dispersed network this assumption holds. Global query optimization is therefore equivalent to minimizing data movements between computers. Binary operations are often translated into *semi-joins* [BER 81.a] to reduce data transfer References [LAF 86], [HUD 85], [CHU 84], [SEG 84], [CHE 84] and [YU 84] all deal with optimizing distributed query processing.

In the dialogue model for distributed query processing, the lack of a global schema inhibits global optimization However, once a host has determined the relations it needs from a neighbour, the pairwise exchange may be optimized. It is possible that in some practical situations the cost of maintaining a global schema offsets the savings generated by global optimization. It will be interesting to implement a system based on the dialogue model and compare its actual performance with that of existing DDBMSs

The work in this thesis will also influence the evolution of the Aldat language Neater formalisms must be found to handle the conversion between data and query representations. The algorithm to convert Aldat expressions to QEXP relations in Chapter 4 will also have to be more precisely defined.

REFERENCES

[AHO 77] Aho A.V., and Ullman J.D. *Principles of Compiler Design*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1977.

[AND 82] Andler S., Ding I., Eswaran K., Hauser C., Kim W., Mehl J. and Williams R. "System D: A Distributed System for Availability", *Proc. Eigth International Conference on Very Large Data Bases*, Mexico City, 1982, pp.33-44.

[BER 81.a] Bernstein P.A. and Chiu D.W "Using Semijoins to Solve Relational Queries", *Journal of the ACM*, Vol.28 No.1, (January 1981) pp.25-40.

[BER 81.b] Bernstein P.A., Goodman N., Wong E., Reeve C.L. and Rothnie J.B. "Query Processing in a System for Distributed Databases (SDD-1)", *ACM Transactions on Database Systems*, Vol.6 No.4, (December 1981) pp.602-625.

[BOR 81] Borr A.J. "Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing", *Seventh International Conference on Very Large Data Bases*, Cannes, France, 1981.

[CER 84] Ceri S and Pelagatti G. *Distributed Databases Principles and Systems*, McGraw-Hill Inc. New York, 1984.

[CHE 84] Chen A.L.P. and Li V.O.K. "Improvement Algorithms for Semijoin Query Processing Programs in Distributed Database Systems", *IEEE Transactions on Computers*, Vol. C-33, No.11, (November 1984) pp.959-967.

[CHU 84] Chung C.W. and Irani K.B. "A Semijoin Strategy for Distributed Query Optimization", *Proc. Fourth International Conference on Distributed Computing Systems*, San Francisco, California, May 1984.

[COD 70] Codd E.F. "A Relational Model of Data for Large Shared Data Banks", *Communications of the ACM*, Vol. 3, No. 6, (June 1970) pp. 377-387.

[COD 72] Codd E.F. "Relational Completeness of Data Base Sublanguages", *Data Base Systems: Courant Computer Science Symposia Series, Vol.6*, Prentice Hall Publishing Co., Englewood Cliffs, N.J., 1972.

[DAT 86] Date C.J. *An Introduction to Database Systems Volume I, Fourth Edition*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1986.

[DAY 85] Dayal U., Buchmann A., Goldhirsch D., Heiler S., Manola F.A., Orenstein J.A. and Rosenthal A.S. *PROBE - A Research Project in Knowledge-Oriented Database Systems: Preliminary Analysis*, CCA-85-02 Computer Corporation of America, Cambribge, Massachusetts, July 1985.

[DEL 80] Delobel C. and Litwin W. *Distributed Data Bases*, North-Holland Publishing Co., Amsterdam, 1980

[FERN 80] Fernandez E.B. *ACM Computing Surveys*, Vol.12, No.1, (March 1980), pp. 111-112.

[FERR 82] Ferrier A. and Stangret C. "Heterogeneity in the Distributed Database Management System SIRIUS-DELTA" *Eighth International Conference on Very Large Data Bases*, Mexico City, 1982, pp.45-53.

[HUD 85] Hudli, A.V. "A New Approach to Distributed Query Processing", *Computer Science and Informatics*, Vol.15, No.1, 1985.

[INR 83] INRIA, France. *Proceedings of Workshop on Relational DBMS Design/Implementation/Use on Microcomputers*, February 1983, Toulouse France.

[KIM 79 Kim W. "Relational Database Systems", *ACM Computing Surveys*, Vol.11, No.3, (September 1979), pp.188-211.

[KOR 86] Korth H.F "Extending the Scope of Relational Languages", *IEEE Software*, Vol.3 No.1, (January 1986), pp.19-28.

[LAF 86, Lafortune S. and Wong E. "A State Transition Model for Distributed Query Processing", *ACM Transactions on Database Systems*, Vol.11 No.3, (September 1986), pp.291-322.

[LAL 86] Laliberté N. *Design and Implementation of a Primary Memory Version of Aldat, Including Recursive Relations*. M.Sc Thesis McGill University School of Computer Science, Montreal, Canada, August 1986.

[MAI 83] Maier D. *The Theory of Relational Databases*, Computer Science Press, Rockville, Maryland, 1983.

[MER 76] Merrett T.H. "The Relational Subalgebra", Course Notes for 308-617 Information Systems, McGill University School of Computer Science, Montreal, Canada, 1976

[MER 84] Merrett T.H. *Relational Information Systems*, Reston Publishing Co, Reston, Virginia, 1984.

[PAG 85] Page T.W., Weinstein M.J. and Popek G.J. "Genesis: A Distributed Database Operating System", *Proceedings of International Conference on Management of Data*, ACM SIGMOD Austin, Texas, 1985, pp 374-387.

[ROT 80] Rothnie J.B., Bernstein P.A , Fox S., Goodman N., Hammer M., Landers T.A., Reeve C., Shipman D.W. and Wong E. "Introduction to a System for Distributed Databases (SDD-1)", *ACM Transactions on Database Systems*, Vol.5 No.1, (March 1980), pp.1-17.

[SEG 84] Segiv A. "Optimizing Fragmented 2-Way Joins", *Proc. Fourth International Conference on Distributed Computing Systems*, San Francisco, California, May 1984.

[SHI 79] Shipman D. "The Functional Data Model and the Data Language DAPLEX", *Proceedings of International Conference on Management of Data*, ACM SIGMOD Boston, MA., 1979.

[SMI 81] Smith J.M., Bernstein P.A., Dayal U., Goodman N., Landers T., Lin K.W.T. and Wong E. "Multibase-Integrating Heterogeneous Distributed Database Systems", *AFIPS Conference Proceedings*, 1981, pp.487-499.

[SMI 77] Smith J.M. and Smith D.C.P. "Database Abstractions: Aggregation and Generalization", *ACM Transactions on Database Systems*, Vol.2 No 2, (June 1977), pp.105-133.

[STO 76] Stonebraker M. and Neuhold E. *A Distributed Data Base Version of INGRES*, Memorandum No. ERL-M612 Engineering Research Laboratory, University of California, Berkeley, September 1976.

[SUZ 82] Suzuki K., Tanaka T. and Hattori F. "Implementation of a Distributed Database Management System for Very Large Real-Time Applications", *Proceedings of Computer Networks COMPCON 82 Twenty-fifth IEEE Computer Society International Conference*, Washington DC, 1982, pp.569-77.

[ULL 82] Ullman J.D. *Principles of Database Systems, Second Edition*, Computer Science Press, Rockville, Md., 1982.

[WEL 81] Welty C. and Stemple D.W. "Human Factors Comparison of a Procedural and a Nonprocedural Query Language", *ACM Transations on Database Systems*, Vol.6, No.4, (December 1981), pp.626-649.

[WIL 82] Williams R., Daniels D., Haas L., Lapis G., Lindsay B., Ng P., Obermarck R., Selinger P., Walker A., Wilms P. and Yost R. "R*: An Overview of the Architecture", *Proceedings of the Second International Conference on Databases: Improving Usability and Responsiveness*, Jerusalem, Israel, 1982.

[YU 84] Yu C.T. and Chang C.C. "Distributed Query Processing", *ACM Computing Surveys*, Vol.16, No.4, (December 1984).