DESIGN AND IMPLEMENTATION OF A HETEROGENEOUS DISTRIBUTED DATABASE MANAGEMENT SYSTEM

Chin-Wan Chung

Computer Science Department
General Motors Research Laboratories
Warren, Michigan 48090-9057

ABSTRACT

This paper presents the architectural design of DATAPLEX, a heterogeneous distributed database management system, and a prototype DATAPLEX. The objective of DATAPLEX is to integrate existing diverse databases as well as emerging new databases within an organization for organization-wide data sharing and data consistency. The architecture of DATAPLEX is based on the relational data model as a common data model and the structured query language (SQL) as a standard query language. A method to decompose distributed queries in this architecture is developed. A prototype DATAPLEX is developed to show the feasibility of the concepts underlying the DATAPLEX approach. The prototype system interfaces an IMS hierarchical database management system (DBMS) on IBM/MVS and an INGRES relational DBMS on DEC/VMS.

1. INTRODUCTION

In a typical data management environment of the manufacturing industry, there are a number of engineering, manufacturing, and business data centers run by various geographically dispersed units. The choices of DBMS's by these data centers are diverse. Currently, there is no effective means to share these heterogeneous databases.

In light of diverse DBMS's and the existence of cooperating autonomous components in an organization, the heterogeneous distributed database system (DDS) is an effective means of sharing data. The heterogeneous DDS can facilitate each data center to select the best DBMS for its environment, because the heterogeneous DDS allows the co-existence of different DBMS's. Although there has been some work on the heterogeneous DDS [3,4,12,13,14], this area is relatively new and there is no proven solution for many technical problems.

DATAPLEX is a heterogeneous distributed database management system (HDDBMS) under development which will allow users and applications to retrieve and update distributed data managed by diverse data systems such that the location of data is transparent to requestors. The key concept of our approach is to use the relational model as a common data model and SQL as a standard query language. This allows us to take advantage of the merits and wide-acceptance of SQL and the relational model and to use results from the research on the homogeneous relational DDS.

In Section 2, we present the architecture of DATAPLEX. This architecture is an open architecture which provides a well-defined interface that can be extended to any database management system and file system. A method for decomposition of distributed queries in this architecture is developed in Section 3.

A prototype system has been developed which interfaces a relational DBMS and a non-relational DBMS using the DATAPLEX approach. The prototype system is described in Section 4. The prototype system processes all the test transactions correctly. The performance of the system is analyzed.

2. ARCHITECTURAL DESIGN

This section presents the architecture of DATAPLEX.

2.1. Heterogeneity Resolution

The architecture of DATAPLEX consists of fourteen functionally independent modules. This architecture is based on the relational model of data [6].

Different data models used by unlike database systems structure data differently. The data definition used by each database system is called the local schema. The data definition of all the sharable databases in the heterogeneous DDS is transformed to an equivalent relational data definition. This common relational data definition is called the conceptual schema. The conceptual schema will include the relational data definition of only the stored data and local views. Each user's view of data in the heterogeneous DDS is called the external schema which consists of a part of the conceptual schema and the views derived from the conceptual schema.

The relational model has been selected for the conceptual schema and the external schema because the relational model provides well-defined query languages and a relational query can be translated to a program in a low-level non-relational data manipulation language (DML) whereas the reverse translation may not be possible.

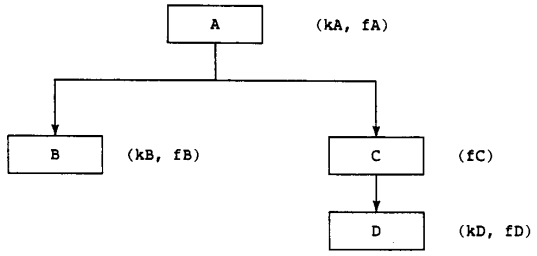
Since we use a common data model for the external schema, a uniform user interface can be provided. Among several relational query languages, we have chosen SQL as the uniform user interface because SQL is widely used and there is a movement to make SQL a standard relational query language.

CH2702-9/89/0000/0356$01.00 © 1989 IEEE

The translation of data definition and DML is necessary to provide the relational model as a common data model and the SQL as a uniform user interface. Our basic method for translating a non-relational data definition to an equivalent relational data definition is the use of the key in the owner record as a foreign key in its member records. This is intended to produce a simple and normalized set of relations. If a record in an access path does not have a key, the descendants of the record do not use any field from the record. This situation is illustrated in the following example:

Example 1

The data structure diagram of a non-relational database is shown in Figure 1, where kX denotes a list of key fields of record X and fX denotes a list of non-key fields of X.



The Data Structure Diagram
of a Non-Relational Data Base

Figure 1

The equivalent relational data definition of the above database is as follows:

RA (kA, fA) RB (kA, kB, fB)
RC (kA, fC) RD (kA, kD, fD)

The underlined list of fields is the key of a relation. In this case, kA in RC is not a key. However, kA in RC can be used to reduce searching in accessing C when a query referencing RC is translated to a non-relational database program.■

The navigation through access paths in non-relational databases can be accomplished by equi-joining relations in an equivalent relational data definition.

Once the common schema is set-up in relational data definitions, users can formulate SQL transactions. External schemas are created by deriving views from the common schema. DATAPLEX must translate an SQL transaction referencing relations in the common schema to a transaction in another relational DML or a non-relational DML. Since the translation among different relational DML's is straightforward, we will only consider the translation to non-relational DML's.

The relational data model and the non-relational data model represent the same information differently. The DML used to formulate requests on information is tightly coupled with the particular representation of the information. Therefore, two basic components for generating a transaction in a particular DML are the semantics of the information request and the data definition which results from data modelling.

In order to translate an SQL transaction to a non-relational database program, the semantics of the information request and the non-relational data definition of the data referenced by the transaction must be obtained. The semantics of the information request can be obtained from the SQL transaction. The semantics include relation names, attribute names, and conditions on attribute values. In addition, access paths in the non-relational data definition can be partially derived from the SQL transaction. This can be achieved by applying an inverse of the data definition translation to the join terms of the SQL transaction because the access path in the non-relational data definition has been translated to joining attributes in the relational data definition.

However, some information about the non-relational data definition is lost in the process of data definition translation. For example, the join term identifies the existence of an access path between the two records, but cannot determine which one is the owner record. Also, an attribute corresponding to a foreign key in a relation may not exist in the corresponding record. This misplaced attribute can also hide an actual access path. Therefore, the information that can be lost in data definition translation must be stored in the translation table and used in the transaction translation.

Transactions will be translated at the location of the computer in which the data referenced by the transaction is stored (data-location) rather than the location of the computer from which the transaction is originated (user-location). The translation at the data-location makes it possible to replicate only the crucial information such as the common schema, security restriction, and the location of data in the DATAPLEX dictionary at every location. (We will not distinguish directory and dictionary here.) The detailed information necessary for transaction translation and query optimization can be stored at the data-location.

2.2. Architecture

The above strategies establish the architecture of DATAPLEX. In a heterogeneous DDS, DATAPLEX is placed with local DBMS at each location. The fourteen modules which constitute the architecture are as follows:

- Controller
- User Interface
- Application Interface
- SQL Parser
- Data Dictionary Manager
- Distributed Transaction Decomposer
- Distributed Database (DDB) Protocol
- Translator
- Local DBMS Interface
- Reformatter

- Distributed Query Optimizer
- Distributed Update Coordinator
- SQL Processor
- Error Handler

The Controller sequences and invokes necessary modules to process a transaction depending on the type of the transaction. The Data Dictionary Manager manages the conceptual schema and the external schema, and finds the location of the data referenced by a transaction. The Distributed Transaction Decomposer decomposes the distributed query into a set of local queries and a user-location query which merges the results from other locations. (A local query references data from a single-location which may be a remote location.) The DDB Protocol interfaces the underlying communication protocol (or medium). It exchanges commands and data with the DDB Protocol of remote DATAPLEX using the file and message transfer facilities of the underlying communication protocol.

The Translator finds transaction translation information from a translation table which records differences of data names and data structures between the conceptual schema and the local schema. The Translator translates an SQL query to a query (or program) in a local DML using the translation information. The Local DBMS Interface sends the translated query to the local DBMS and obtains the local result. The Reformatter loads a file (e.g., local result) as a temporary relation and set up necessary indices when further relational data manipulations are necessary.

The Distributed Query Optimizer of the user-location DATAPLEX (source DATAPLEX) schedules an optimal data reduction plan using the statistical information from the data-location DATAPLEX's (target DATAPLEX's). Our data reduction plan [5] is a sequence of semijoins which consists of local data reduction operations and data moves among computers. The Distributed Query Optimizer executes the data reduction plan by sending commands for data reduction operations and data moves to target DATAPLEX's. The Distributed Update Coordinator performs distributed concurrency control and distributed data recovery. The SQL Processor is capable of processing SQL queries for merging local results or further manipulating intermediate results locally. The functions of other modules are self-explanatory.

The way of packaging the above modules into processes depends on the detailed design consideration. In this section, we assume that DATAPLEX is one process for simplicity. DATAPLEX's at the user-location and the data-location(s) communicate through the DDB protocol. The modules Translator and Local DBMS Interface are not used at the user-location unless the user-location is the same as the data-location.

This architecture is independent of the local data system except for the modules Translator and Local DBMS Interface. Any data systems can be interfaced to DATAPLEX by developing these two modules for them. Also, different communication protocols can be used by adapting the DDB Protocol to them.

3. DISTRIBUTED QUERY DECOMPOSITION

In this section, we present our distributed query decomposition method and explain its relevance in a heterogeneous DDS.

3.1. The Method

A distributed query cross-references data stored at more than one location. The decomposition requires information on the locations of referenced data. This information will be stored in the DATAPLEX dictionary. The result of our decomposition is a set of local queries and a user-location query which merges local results. Since the local queries do not depend on each other, they can be optimized and executed in parallel. If there are multiple data systems in a computer, a local query to this computer has to be further decomposed into a set of queries each of which references data from a data system. In this case, the location becomes the system identification of a data system. Our query decomposition method will also be used for this case without any modification.

There are two types of relational query languages, the relational algebra language and the relational calculus language. The relational calculus language is semantically higher than the relational algebra language. Our query decomposition generates local queries in a relational calculus language, specifically in SQL, for the following reasons:

- The relational calculus language is more widely used among relational DBMS's. Therefore, generating local queries in SQL will simplify the query translation.

- A typical sequence of relational operations formulated to process a request cannot in general be translated to a program in a non-relational DML in this sequence. It is necessary to send a query in a semantically high language to a non-relational DBMS.

The subqueries are generated in a textual form rather than other equivalent representations used by some relational DDBMS's. This is an important consideration in a heterogeneous DDS because the translation is basically the text manipulation and also some of the major non-relational DBMS's will provide an SQL interface soon.

The predicate (WHERE clause) of an SQL query contains Boolean operators AND, OR, and NOT. In addition, parentheses can be used to specify the precedence in the predicate. If the precedence of Boolean operators is not correctly handled, an erroneous result is produced. Since the precedence is difficult to handle for distributed queries, we decompose a query into conjunctive queries containing only AND operators as follows:

1. NOT's can be eliminated by using DeMorgan's Law and negating relational operators such as =, ¬= and >.

2. OR's and parentheses can be eliminated by transforming the predicate into a disjunctive

normal form and processing each conjunctive term as a separate query. The result of the original query is the union of the results produced by the conjunctive sub-queries.

SQL allows nesting of a query within another query. A nested query can be processed as a sequence of non-nested queries by processing the inner queries in the nest first. Also, the nesting can be eliminated by transforming a nested query to an equivalent non-nested query [11].

From the above discussion, the basic part of a distributed query is a conjunctive query in the following form:

```
SELECT list of target attributes and functions
FROM   list of relations
WHERE  qualification
aggregate clause;
```

where functions are expressions and set functions COUNT, SUM, AVERAGE, MAX, and MIN; qualification contains selection terms and join terms; and aggregate clause contains GROUP BY, HAVING, and ORDER BY.

The following variables are defined to explain our algorithm:

- R(i): the ith relation referenced by a query
- ATR(Ri): the set of all the attributes of the relation R(i)
- A(Ri): a subset of ATR(Ri)
- T: the set of target attributes of a query
- G: the set of attributes in the aggregate clause and functions. If a query contains COUNT(*), G includes all the attributes in R(i)'s which are not in selection terms or in join terms
- A(LJ): the set of local joining attributes
- A(GJ): the set of global joining attributes

Our distributed query decomposition algorithm, QUERY_DECOMP, is as follows:

ALGORITHM QUERY_DECOMP

Input: A distributed conjunctive SQL query and the locations of referenced relations

Output: A set of local SQL queries and a user-location SQL query

Step 1. Initialize R(i), ATR(Ri), T, and G.

Step 2. Assign the SELECT list and the aggregate clause of the input query to those of the user-location query.

Step 3. Transform the input query to a query graph Q with relations as nodes and qualification terms as edges.

Step 4. For each node R(i), associate an attribute set A(Ri) such that A(Ri) ← ATR(Ri) ∩ (T U G), where ∩ is the set intersection and U is the set union.

Step 5. Using the location information,

(1) Delete the edges corresponding to global join terms from Q. Q is decomposed into connected sub-graphs Q(j)'s.
(2) Assign global join terms to the qualification of the user-location query.
(3) Determine A(LJ) and A(GJ).
(4) A(Ri) ← A(Ri) U {ATR(Ri) ∩ A(GJ)}

Step 6. Transform each Q(j) to an SQL query as follows:
(1) target attribute set ← U over i A(Ri) for R(i)'s in Q(j)
(2) FROM list ← R(i)'s in Q(j)
(3) qualification ← selection terms and join terms corresponding to edges in Q(j)

Let L(j) be the result relation of the local query transformed from Q(j).

Step 7. Assign L(j)'s to the FROM list of the user-location query. For each R(i) in the user-location query, replace R(i) with L(j) corresponding to Q(j) which contains R(i).

Local queries are generated in step 6 and the user-location query is formed in steps 2, 5 and 7. The following theorem provides the correctness of the algorithm.

Theorem 1: The queries generated in step 6 of QUERY_DECOMP are local and contain all necessary target attributes. The execution of the local queries and the user-location query produces the correct result for the original distributed query.

Proof: The target attributes of the local query corresponding to Q(j) should be the attributes of R(i)'s in Q(j) which are also in T U G U A(GJ), that is {U over i ATR(Ri) for R(i)'s in Q(j)} ∩{T U G U A(GJ)}. Let X = T U G U A(GJ). Since {U over i ATR(Ri)} ∩ X = U over i {ATR(Ri) ∩ X}, which can be proved by mathematical induction, step 6. (1) generates correct target attributes. A query q can be processed by a sequence of relational operations. One correct sequence $\sigma 1$ is S, J, P, where S is a sequence of all necessary selections, J is a sequence of all necessary joins, and P is a sequence of all necessary projections. Functions and the aggregate clause are locally processed for both local queries and distributed queries. If q is distributed, the decomposition of q by QUERY_DECOMP corresponds to a particular sequence $\sigma 2$ for processing q. QUERY_DECOMP partitions J into local joins J(L) and global joins J(G). In addition, QUERY_DECOMP schedules intermediate projections P(I) between J(L) and J(G). In other words, $\sigma 2$ = S, J(L), P(I), J(G), P. However, all the joining attributes for J(G) are retained after P(I). Therefore, P(I) and J(G) commute. P(I), P = P because the attributes used for P are contained in those for P(I). Consequently, $\sigma 2$ = S, J(L), P(I), J(G), P = S, J(L), J(G), P(I), P = S, J, P = $\sigma 1$. ∎

3.2. An Example

The following example illustrates the steps in QUERY_DECOMP. The distribution of relations and a

distributed query are as follows:

Relations (and their attributes) and locations:

```
SUPPLIER (S#, SNAME, STATE).....LOC1
PART     (P#, PNAME, MTRL).....LOC2
SUPPLY   (S#, P#, QTY).....LOC2
```

A distributed query:

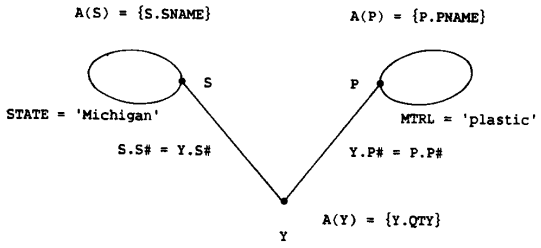
```
SELECT  SNAME, SUM (QTY)
FROM    SUPPLIER, SUPPLY, PART
WHERE   STATE = 'Michigan'
AND     SUPPLIER.S# = SUPPLY.S#
AND     SUPPLY.P# = PART.P#
AND     MTRL = 'plastic'
GROUP   BY SNAME
HAVING  COUNT (*) > 5;
```

The verbal description of the query is "for suppliers in Michigan which supply more than five different plastic parts, find the names of the suppliers and the total quantity of plastic parts supplied by each supplier". The execution steps of QUERY_DECOMP are as follows:

1. Let S be the relation SUPPLIER, P the relation PART, and Y the relation SUPPLY. Then
   ATR(S) = {S.S#, S.SNAME, S.STATE},
   ATR(P) = {P.P#, P.PNAME, P.MTRL},
   ATR(Y) = {Y.S#, Y.P#, Y.QTY}.
   From the input query,
   T = {S.SNAME},
   G = {Y.QTY, S.SNAME, P.PNAME}.

2. Construct an intermediate form of the user-location query as follows:
   SELECT SNAME, SUM (QTY)
   GROUP BY SNAME
   HAVING COUNT (*) > 5;

3. A query graph Q corresponding to the relations S, P, Y and the qualification terms of the input query is shown in Figure 2.



The Query Graph Q
with Initial Attribute Sets

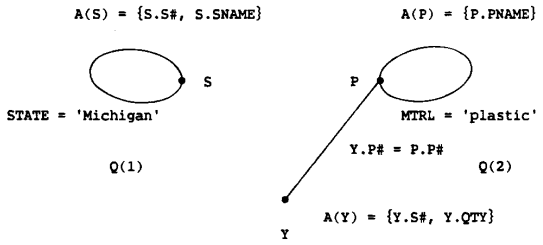
Figure 2

4. T U G = {Y.QTY, S.SNAME, P.PNAME}. For each node in Q, an attribute set is defined as follows:

```
A(S) = ATR(S) ∩ (T U G)
     = {S.S#, S.SNAME, S.STATE} ∩{Y.QTY,
       S.SNAME, P.PNAME}
     = {S.SNAME}
A(P) = ATR(P) ∩ (T U G)
     = {P.PNAME}
A(Y) = ATR(Y) ∩ (T U G)
     = {Y.QTY}
```

5. (1) Since S is at LOC1 while P and Y are at LOC2, the edge S.S# = Y.S# is deleted from Q.
   (2) An intermediate form of the user-location query becomes
   ```
   SELECT  SNAME, SUM (QTY)
   WHERE   S.S# = Y.S#
   GROUP   BY SNAME
   HAVING  COUNT (*) > 5;
   ```
   (3) Since S.S# = Y.S# is a global join term,
   A(GJ) = {S.S#, Y.S#} and
   A(LJ) = {Y.P#, P.P#}.
   (4)
   ```
   A(S)  ← A(S) U {ATR(S) ∩ A(GJ)}
         = {S.SNAME} U {S.S#}
         = {S.SNAME, S.S#}
   A(P)  ← {P.PNAME} U Null-set
         = {P.PNAME}
   A(Y)  ← {Y.QTY} U {Y.S#}
         = {Y.QTY, Y.S#}
   ```

   The decomposed query graph Q(j)'s and an updated attribute set for each relation are shown in Figure 3.



The Decomposed Query Graph
with Updated Attribute Sets

Figure 3

6. For each Q(j), construct a local query and assign the name of the local result relation L(j) as follows:

   ```
   L(1) = SELECT S#, SNAME
          FROM   S
          WHERE  STATE = 'Michigan';

   L(2) = SELECT PNAME, S#, QTY
          FROM   P, Y
          WHERE  MTRL = 'plastic'
          AND    Y.P# = P.P#;
   ```

7. Since S is at LOC1 and P and Y are at LOC2, the user-location query becomes as follows:

```
SELECT  SNAME, SUM (QTY)
FROM    L(1), L(2)
WHERE   L(1).S# = L(2).S#
GROUP   BY SNAME
HAVING  COUNT (*) > 5;
```

4. PROTOTYPE IMPLEMENTATION

In this section, we present the implementation of a prototype DATAPLEX and our experience with the prototype system.

The prototype system interfaces IMS hierarchical DBMS on IBM/MVS and INGRES relational DBMS on DEC/VMS. The prototype system allows users to retrieve data from IMS and/or INGRES with a single SQL query from DEC/VMS such that the location of data is transparent to requestors. The unique features that the prototype system provides to users are as follows:

(1) SQL queries to IMS
(2) Distributed SQL queries to IMS and INGRES
(3) Distributed SQL queries embedded in a C language program

The data types supported between IBM and DEC computers are: characters, text (variable length fields), integers, floating point numbers, and packed decimal numbers. The prototype system checks whether a user is authorized to access IMS data at a segment level using the userid.

However, since rapid prototyping was required to show the feasibility before developing a full-function DATAPLEX, the update of IMS data and the full query optimization were not implemented in the prototype system. In addition, the system supports a subset of SQL defined to have the following syntax:

```
SELECT    list of target attributes and set
          functions
FROM      list of relations
WHERE     qualification
ORDER BY  attributes
```

Where set functions are MAX, MIN, SUM, COUNT, AVERAGE, and the qualification contains >, >=, <, <=, =, ¬=, AND, OR, NOT and parentheses.

The core of the prototype system was structured in four processes as follows:

(1) User Interface (UI)

This process corresponds to the DATAPLEX modules: User Interface and Application Interface.

(2) Distributed Query Manager (DQM)

This process corresponds to the DATAPLEX modules: SQL Parser, Data Dictionary Manager, Distributed Transaction Decomposer, and a part of Distributed Query Optimizer.

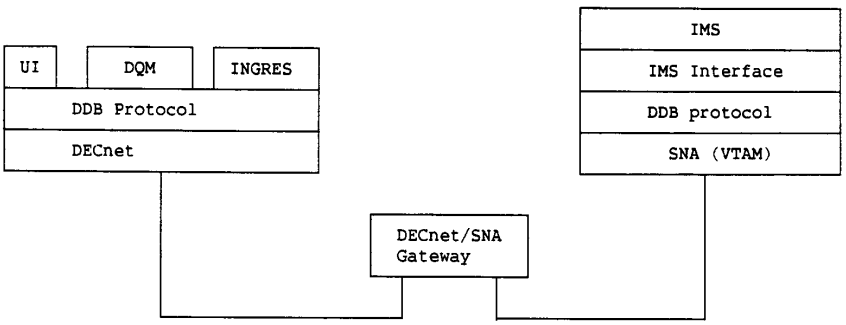
(3) IMS Interface

This process corresponds to the DATAPLEX modules: Translator and Local DBMS Interface.

(4) DDB Protocol

Since IMS runs on IBM/MVS and INGRES runs on DEC/VMS, communication between MVS and VMS must be provided. A DECnet/SNA GATEWAY [8] from DEC was used to connect IBM's Systems Network Architecture (SNA) network and DEC's DECnet. DDB Protocol supports the program to program communication between DQM and IMS Interface by using the functions provided by DECnet and SNA network.

The overall architecture of the prototype system is shown in Figure 4. On a DEC computer, DDB Protocol provides communication among the processes of UI, DQM, and INGRES. DDB protocol also provides the interface via DECnet and the DECnet/SNA GATEWAY to the DDB Protocol running on the IBM side. Some modules on IBM/MVS were implemented in IBM assembler. C language was used to implement all other modules.



The Architecture of DATAPLEX Prototype

Figure 4

A test bed has been established at General Motors Research Laboratories. IMS is installed on an IBM 4381 MODEL 14. INGRES is installed on a DEC VAX 11/785. Users run the prototype system through the VAX/VMS. A test distributed database and test transactions were created. Users formulate requests based on the relational view of the distributed database. The location and the type of the actual database are transparent to users. The prototype system processes all the test transactions correctly.

Production IMS data is used to test the effect of the size of the database on efficiency. The data is from the Maintenance Management Information System (MMIS) running at a car assembly plant. The size of the MMIS database is about 1,000 times bigger than that of the test IMS database. SQL queries are executed against the MMIS database and the requests formulated as the SQL queries were also programmed in PL/I. The result shows that the prototype system incurs some overhead compared with the access using PL/I programs. As the database size grows, the fraction of the overhead to the total processing time decreases. In retrieving data from an IMS database containing 87,000 records which is a part of the MMIS database, the average response time (the clock time) of an SQL query is 368 seconds whereas the corresponding PL/I program takes 306 seconds.

5. SUMMARY

The architecture of DATAPLEX is modular and is an open architecture which provides facilities for easy interfaces to additional data systems.

A distributed query decomposition method suitable for a heterogeneous distributed database system is developed. Three of the important technical problems which need more research are:

(1) Distributed concurrency control and distributed data recovery in a heterogeneous distributed database system.
(2) Efficient translation of an SQL query to an optimal program to access non-relational databases.
(3) Distributed data dictionary management including global naming procedures and interfaces to local data dictionaries.

A DATAPLEX prototype has been developed which interfaces an IMS hierarchical DBMS on IBM/MVS and an INGRES relational DBMS on DEC/VMS. The capability of the prototype system proves the feasibility of the concepts underlying the DATAPLEX approach to solving the problem of transparently sharing data in a diverse database environment.

ACKNOWLEDGMENT

The prototype system was jointly developed with Relational Technology, Inc.

REFERENCES

[1] Apres, P. M., Hevner, A. R., and B. Yao, "Optimization algorithms for distributed queries," IEEE TSE, Vol. SE-9, January 1983, pp. 57-68.
[2] Bernstein, P. A. and D. M. Chiu, "Using semi-joins to solve relational queries," the Journal of ACM, January 1981, pp. 25-40.
[3] Breitbart, Y. J. and L. R. Hartweg, "RIM as an implementation tool for a distributed heterogeneous database," Proc. of IPAD II Symposium on Advances in Distributed Data Base Management for CAD/CAM, April 1984, pp. 155-164.
[4] Cardenas, A. F. and M. H. Pirahesh, "Database communication in a heterogeneous database management system network," Inform Systems, Vol. 5, 1980, pp. 55-79.
[5] Chung, C. W. and K. B. Irani, "An optimization of queries in distributed database systems," the Journal of Parallel and Distributed Computing, Vol. 3, No. 2, June 1986, pp. 137-157.
[6] Codd, E. F., "A relational model of data for large shared data banks," Communications of the ACM, Vol. 13, June 1970, pp. 377-387.
[7] Dayal, U. and N. Goodman, "Query optimization for CODASYL database systems," Proc. of the ACM SIGMOD Conference, June 1982, pp. 138-150.
[8] Digital Equipment Corporation, Networks and Communications Buyers Guide, 1986.
[9] Elmagarmid, A. K., "A survey of distributed deadlock detection algorithms," ACM SIGMOD Record, September 1986, pp. 37-45.
[10] Eswaran, K. P. et al., "The notions of consistency and predicate locks in a database system," Communications of the ACM, Vol. 19, November 1976, pp. 624-633.
[11] Kim, W., "On optimizing an SQL-like nested query," ACM TODS, Vol. 7, No. 3, September 1982, pp. 443-469.
[12] Landers, T. and R. L. Rosenberg, "An overview of Multibase," Distributed Data Bases, North-Holland, 1982, pp. 153-184.
[13] Su, S. Y. W. et al., "The architecture and prototype implementation of an Integrated Manufacturing Database Administration System," Spring COMPCON, 1986.
[14] Takizawa, M., "Heterogeneous distributed database system: JDDBS," Database Engineering, Vol. 6, No. 1, March 1983, pp. 58-62.
[15] Yu, C. T. et al., "Query processing in a fragmented relational distributed system: Mermaid," IEEE TSE, Vol. SE-11, August 1985, pp. 795-810.
[16] Zaniolo, C., "Design of relational views over network schemas," Proc. of the ACM SIGMOD Conference, May 1979, pp. 179-190.