Heterogeneous Distributed Database Systems for Production Use

GOMER THOMAS

Bellcore, 444 Hoes Lane, Piscataway, NJ 08854

GLENN R. THOMPSON

Amoco Production Company Research Center, 4502 East 41st Street, Tulsa, Oklahoma 74135

CHIN-WAN CHUNG

Computer Science Department, GM Research Laboratories, Warren, Michigan 48090

EDWARD BARKMEYER

National Institute of Standards and Technology, Gaithersburg, Maryland 20899

FRED CARTER

Ingres Corporation, 1080 Marina Village Parkway, Alameda, California 94501

MARJORIE TEMPLETON

Data Integration, Inc., 3233 Federal Avenue, Los Angeles, California 90066

STEPHEN FOX

Xerox Custom Systems Division, 7900 Westpark Drive, McLean, Virginia 22102

BERL HARTMAN

Sybase, Inc., 6475 Christie Avenue, Emeryville, California 94608

It is increasingly important for organizations to achieve additional coordination of diverse computerized operations. To do so, it is necessary to have database systems that can operate over a distributed network and can encompass a heterogeneous mix of computers, operating systems, communications links, and local database management systems. This paper outlines approaches to various aspects of heterogeneous distributed data management and describes the characteristics and architectures of seven existing heterogeneous distributed database systems developed for production use. The objective is a survey of the state of the art in systems targeted for production environments as opposed to research prototypes.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed databases*; H.2.4 [**Database Management**]: Systems; H.2.5 [**Database Management**]: Heterogeneous Databases

General Terms: Design

Additional Key Words and Phrases: Database integration, distributed query management, distributed transaction management, federated database, multidatabase, system architecture

---

CONTENTS

INTRODUCTION

The objectives of this survey paper are to provide insight into the kinds of heterogeneous distributed database capabilities available in off-the-shelf systems and to describe the practicalities of in-house development of their capabilities; that is, what can be achieved and what approaches are likely to be promising.

This paper takes a broad view of the terms "distributed", "heterogeneous", and "production use." A database system is considered to be *distributed* if it provides access to data located at multiple (local) sites in a network, even if it does not provide full facilities for schema integration, distributed query management, and/or distributed transaction management. It is considered to be *heterogeneous* if the local nodes have different types of computers and operating systems, even if all local databases are based on the same data model and perhaps even the same database management system. (This definition of heterogeneous is admittedly at odds with that commonly used by the research community, where the term is typically used to imply multiple data models.) It is considered to be for *production use* if it has been, or is being, developed for that purpose, even if it is currently only in an advanced prototype state.

Although a number of specific database systems are described or mentioned in this paper, this paper is not intended to contain a definitive list of heterogeneous distributed database systems for production use or to endorse or recommend any particular system. The information about specific systems may or may not be entirely accurate, and it is likely that other systems on the market, in production use, or under development have capabilities equaling or exceeding those mentioned here.

Section 1 discusses different types of distributed database capabilities that can be provided by different systems. Section 2 describes the status of remote database access standards, a key factor in the ease of developing heterogeneous distributed database systems. Section 3 describes the characteristics and architecture of a sampling of existing heterogeneous distributed database systems developed for production use. These descriptions are current as of late 1989, written by individuals who have been involved in the development of these systems. Section 4 gives a brief summary of the state of the art.

1. HETEROGENEOUS DISTRIBUTED DATABASE CAPABILITIES

Different types of capabilities can be provided by heterogeneous distributed database systems. They include schema integration, distributed query processing, distributed transaction management, administrative functions, and coping with different types of heterogeneity. Schema integration has to do with the way in which users can logically view the distributed data. Distributed query management deals with the analysis, optimization, and execution of queries that reference distributed data. Distributed transaction management deals with the atomicity, isolation, and durability of transactions in a distributed system. Administrative functions include

such things as authentication and authorization, defining and enforcing semantic constraints on the data, and management of data dictionaries and directories. Heterogeneity can include differences in hardware, operating systems, communications links, database management system (DBMS) vendors, and/or data models. These are all important aspects of distributed data management. In considering them, it is important to recognize there is no "ideal" set of capabilities for all environments or applications. A particular capability may be invaluable in certain situations while being totally unsuitable in others.

1.1 Schema Integration

Each local database has a local schema, describing the structure of the data in that database. Each user has a user view, describing that portion of the distributed data that is of interest to the user, possibly reorganized to meet the user's requirements. There are two general approaches to the problem of providing mappings between user views and the local schemata:

(1) All of the local schemata may be integrated into a single *global schema* that represents all data in the entire distributed system. All user views are derived from this global schema.

(2) Various portions of various local schemata may be integrated into multiple *federated schemata.* These federated schemata may correspond closely to user views, or user views may be further derived from these federated schemata.

In either approach one is faced with schema integration—the process of developing a conceptual schema that encompasses a collection of local schemata.

If the local databases are based on different data models and/or use different names or representations for particular data elements, the usual first step is for each local schema to be translated into an equivalent schema in some common data model using common names and representations. Thus, in this case one is also faced with a schema translation problem.

The simplest form of schema integration is *schema union,* whereby a conceptual schema seen by distributed users is the disjoint union of local schemata or perhaps of subsets or views of local schemata. In the simplest form of this, a data item in the conceptual schema is identified by the name of a local database concatenated with the name of an item in the local database. More commonly, some type of aliasing capability is provided.

More sophisticated capabilities that can be provided include the following:

- *Replication,* whereby data items in different local databases may be identified as copies of each other
- *Horizontal fragmentation,* whereby data items in different local databases may be identified as logically belonging to the same table or entity set in the integrated schema
- *Vertical fragmentation,* whereby data items in different local databases may be identified as logically representing the same row or entity in the integrated schema but containing different attributes for the row or entity
- *Data mapping,* whereby data types or data values are converted for conformity with each other

As an example of data mapping, one local database might store temperatures as floating point numbers representing degrees Celsius, and another local database might store temperatures as character string representations of degrees Fahrenheit. To integrate these schemas, one might want to map degrees Fahrenheit to degrees Celsius and map character string representations of numbers to floating point representations, or vice versa.

1.2 Distributed Query Management

Distributed query management provides the ability to combine data from different local databases in a single retrieval operation (as seen by the user). This capability may or may not be provided by a distributed database system. In some systems, an application needing data from multiple local

databases would explicitly send queries to the individual databases rather than presenting a single distributed query to a distributed query manager. If distributed query management is provided, there are many possible approaches to optimizing the heterogeneous distributed queries and managing their execution. To complicate the optimization problem, there may be great differences in the speeds of the communications links, the speeds and workloads of the local processors, and the nature of the data operations available at the local sites. Depending on the system characteristics, it may be desirable to exploit parallelism in the execution of queries. Replicated and/or fragmented data also add complexity. One of the simpler approaches to query management is to move all the desired data to the query site and combine them there. More sophisticated approaches may consider a wide range of algorithms and take account a wide range of factors in evaluating them [Chen et al. 1989].

1.3 Distributed Transaction Management

Distributed transaction management provides the ability to read and/or update data at multiple sites within a single transaction, preserving the transaction properties of atomicity, isolation, and durability [Ceri and Pelagatti 1984]. This capability may or may not be provided by a distributed database system. If it is provided, there are two aspects provided: concurrency control protocols and commit protocols.

Local concurrency control protocols ensure that the execution of local transactions, whether originating at the local database or coming from a remote site, meet the isolation standards of the local system. In most cases this means the execution is serializable; that is, the actual execution is equivalent to the serial execution of the transactions in some order [Eswaran et al. 1976]. Distributed concurrency control protocols are designed to ensure that distributed transactions are globally serializable; that is, that the serializations of the local components at all the databases are compatible with some global serialization of all the distributed transactions [Traiger et al. 1982].

Local commit protocols guarantee atomicity and durability of local transactions; that is, that either all of the actions of a local transaction complete and commit or none of them do. Distributed commit protocols guarantee global atomicity and durability; that is, that either all of the local subtransactions of a distributed transaction complete and commit or none of them do.

These distributed protocols carry an operational cost, as well as an implementation cost, since a failure at one site may leave locked data at other sites unavailable for extended periods of time. Thus, there can be advantages to not having them if they are not needed. In some situations the semantics of the databases and applications may make both distributed concurrency control and distributed commit protocols unnecessary. An example of this is a collection of independently maintained reference databases (e.g., restaurant ratings or bibliographic listings), with a capability for users to submit queries that form the union of information from the different databases in the collection. Since the databases are being independently updated and since users are typically interested in partial information, even if one of the databases is temporarily unavailable, there is no need to ensure global serializability or global atomicity. In other situations, distributed commit protocols, but not distributed concurrency control, may be needed. An example of this is a travel reservation system in which reservations for different resources are independently managed in different databases. The order in which reservations for different resources are interleaved is not critical, so distributed concurrency control is not necessary. A user would, however, often want to package a number of interdependent reservations into a single transaction and require that either they all commit or none of them do.

In other situations, both distributed concurrency control and commit protocols may be needed. An example is a banking system in which accounts at different branches are maintained in different databases. Global

atomicity is needed to ensure that funds transfers between branches are handled correctly. Global serializability is needed to ensure that audit programs run correctly, even when run concurrently with funds transfer programs.

1.4 Administration

A number of administrative functions, such as authorization of users, definition and enforcement of semantic integrity constraints on the data, and maintenance of schema information, must be performed for any database system. In a heterogeneous distributed database system, there are many choices as to the degree of centralization and transparency of these administrative functions.

1.4.1 Authorization Management

At one extreme, authorization can be completely decentralized, with each user having a user id at each local database and permissions granted by local database adminstrators and enforced by local database management systems. At the other extreme, authorization can be completely centralized, with each user having a system-wide user id and permissions granted on a system-wide basis by a system-wide database administrator and enforced by a system-wide distributed query manager. An intermediate possibility is to have system-wide user ids but with permissions granted and enforced at local databases.

One potential advantage of centralized permissions is that a user can be given access to certain distributed views while being denied direct access to the underlying local data. For example, one might have a table in one database that tells which employees are assigned to which division of a company and another table in another database that gives the salary of each employee. One might want certain users to have access to the average salaries in the different divisions of the company but not to the salaries of individual employees. With centralized permissions, it is straightforward to define a view containing the average salary information (obtained by taking the join of the two local tables and applying aggregation and projection operations) give the users permission to access the view but not permission to access the underlying employee salary table. With only local permissions, the only choices are to give the user access to the employee salary table or not to. There is no mechanism for preventing access to the salary table but granting access to a view derived from a join of the salary table with a table in another database.

1.4.2 Semantic Integrity Management

Some database management systems provide capabilities for specifying and enforcing semantic integrity rules for the database instead of leaving that function to the applications programmers. These include such things as rules for allowed data values and existence dependencies. In the context of heterogeneous distributed databases, this can be done either centrally, through a global conceptual schema, or locally, through the individual local schemas. One potential advantage of the centralized approach is that distributed integrity rules, that is, rules that depend on data stored in different databases, can be enforced.

1.4.3 Location Transparency and Schema Maintenance

Location transparency is the ability to deal with distributed data without having to know where it is or even that it is distributed. There are two levels at which location transparency is relevant: the level of the user (programmer or interactive user) and the level of the database administrator.

At one extreme, users may have to specify data items by machine name or address, DBMS identifier, database name, and data item name. At the other extreme, users may only have to specify logical names of data items. For pure logical data item naming to work, there must be some kind of naming convention that guarantees uniqueness of names for all data items accessible by the user, a troublesome requirement if there are users who need access to essentially all the data of the enterprise. A compromise is

to require the user to specify a logical database name and a data item name, with the system mapping logical database names to local databases or federated schemata. This requires only system-wide uniqueness of logical database names and uniqueness of data item names within each logical database.

There is also a range of choices for how the database administrator(s) provide location transparency for the programmer or interactive user. At one extreme, the database administrator(s) may individually maintain multiple data dictionaries at multiple sites describing what data can be found where. These data dictionaries may all represent a global schema and describe all the data in the system, or they may each represent a particular federated schema and only describe data of interest to a particular group of users. At the other extreme, the system itself may make all decisions about data placement and maintain all directories automatically (although this extreme would be highly unusual in a heterogeneous system). A compromise approach is to have any changes in data placement be entered only once and to have the system automatically propagate the information throughout the system as needed.

In a system with multiple federated schemata, the data dictionaries and directories are typically physically decentralized. In a system with a single global schema, they are logically centralized but may be physically centralized or distributed. In either case, as indicated above, varying degrees of centralization and coordination are possible for their maintenance.

1.5 Types of Heterogeneity

Many people in the research community have traditionally regarded a "heterogeneous" database system as one involving multiple data models. From a practical standpoint, however, a distributed database system may involve a number of different types of heterogeneity—computer hardware, operating systems, communications links and protocols, database management systems (vendors), and/or data models—each of which presents its own problems to the system developer. Different hardware and operating systems may use different data representations, for example, different character codes or different representations for floating point numbers. Different communications mechanisms require gateways, and these can be difficult to implement because of the different capabilities involved. For example, one protocol may allow an out-of-band interruption of a session, whereas another may not. Different DBMS vendors may use different data definition and data manipulation languages, even though they may be using the same data model. Different data models can present a difficult schema translation and query translation problem, especially if performance is important.

2. STANDARDS ACTIVITIES

One key to efficient development of heterogeneous distributed database systems is a standard language and protocol for remote data access. If all of the local sites in the system supported a common set of data operations, accessible via a standard language, it would greatly simplify managing distributed queries. If support for a "prepare to commit" state was also standardized, implementing a distributed two-phase commit protocol would be very straightforward. Moreover, if the local systems all used strict two-phase locking (locks held until commit point), as many of today's database systems do, distributed concurrency control would be implied by distributed two-phase commit [Breitbart and Silberschatz 1988]. Distributed deadlock detection would, however, be needed.

There are two major hurdles to overcome in reaching a consensus on the many technical issues involved in such standards. One is the usual problem of getting a diverse community of vendors and users to agree on a common way of doing things when they may already have investments in separate solutions. The other problem, which in this case, is perhaps more serious is that there has still been relatively little experience with distributed databases in production use. Even those organizations that have implemented them often do not feel

they have a good understanding of precisely what functionality should be provided, let alone precisely how various things should be handled in the remote data access language. There is not even a widely accepted reference model for distributed database systems.

Nonetheless, both the International Standards Organization (ISO) and the American National Standards Institute (ANSI) are active in this area. The RDA Rapporteur Group of Working Group 3 on Database of ISO TC97/SC21 was formed in late 1985 to work on a Remote Data Access (RDA) standard, and Subcommittee X3H2.1 of Technical Committee X3H2 on Databases of the ANSI/X3 Committee on Information Processing Systems was formed in late 1988 to coordinate U.S. input into the ISO process. It is possible that an ISO Draft International Standard could be approved as early as mid-1990 and an International Standard could be approved as early as mid-1991.

3. SOME EXISTING SYSTEMS

This section presents a brief description of a sampling of systems that have been developed or are being developed for production use. Four general types of information are given for each system:

(1) Background—motivation, objectives, and history for the project or product
(2) System characteristics—capabilities or features of the system, including its positioning in Sheth and Larson's taxonomy of heterogeneous distributed database systems [Sheth and Larson 1990]
(3) System architecture—major system components and their functions
(4) Current status and future plans—including, in some cases, an estimate of the amount of work that has gone into the project or product and/or some of the major lessons learned during development

To avoid numerous footnotes, we consolidated the following trademark acknowledgements: IBM, IMS, SQL/DS, DB2, IBM PC, OS/2, VM, CMS, MVS, and SNA are trademarks of International Business Machines Corporation. DEC, VMS, and VAX are trademarks of Digital Equipment Corporation. Sun Workstation is a trademark of Sun Microsystems, Inc. Apollo Domain is a trademark of Apollo Computers, Inc. RIM is a trademark of Boeing Computer Services. FOCUS is a trademark of Information Builders, Inc. UNIX is a registered trademark of AT&T. ORACLE is a trademark of Oracle Corporation. IDM is a trademark of Sharebase, Inc. Mermaid is a trademark of UNISYS Corporation. System 2000 is a trademark of Intel Corporation. Ingres/STAR and Ingres/Gateway are trademarks of Ingres Corporation (formerly Relational Technology, Inc.). SQL Server, SQL Toolset, Open Client, and Open Server are trademarks of Sybase, Inc.

3.1 ADDS (Amoco Production Company, Research)

3.1.1 Background on ADDS

The Amoco Distributed Database System (ADDS) [Breitbart and Tieman 1985; Breitbart et al. 1986] project began in late 1983, responding to the problem of integrating databases distributed throughout the corporation. Applications were being planned that required data from multiple sources. At the time, database products did not provide effective means for accessing or managing data from diverse systems. Therefore, the ADDS project was initiated to simplify distributed data access and management within Amoco.

3.1.2 ADDS System Characteristics

ADDS provides uniform access to preexisting heterogeneous distributed databases. The ADDS system is based on the relational data model and uses an extended relational algebra query language. A subset of the ANSI SQL language standard is also supported. In the terminology of [Sheth and Larson 1990], ADDS is a tightly coupled federated system supporting multiple

federated schemata. Local database schemata are mapped into multiple federated database schemata, called Composite DataBase (CDB) definitions. The mappings are stored in the ADDS data dictionary. The data dictionary is fully replicated at all ADDS sites to expedite query processing. A CDB is usually defined for each application. Multiple applications and users may, however, share CDB definitions. Users must be authorized to access specific CDBs and relational views that are defined against the CDBs.

The CDBs support the integration of the hierarchical, relational, and network data models. Local DBMSs currently supported include IMS, SQL/DS, DB2, RIM, INGRES, and FOCUS. Semantically equivalent data items from different local databases, as well as appropriate data conversion for the data items, may be defined.

The user interface consists of an Application Program Interface (API) and an interactive interface [Lee et al. 1988]. The API consists of a set of callable procedures that provide access to the ADDS system for application programs. Programs use the API to submit queries for execution, access the schema of retrieved data, and access retrieved data on a row-by-row basis. The API provides programmers with location and DBMS transparent access to distributed databases.

The interactive interface allows terminal users to execute queries, display the results of the queries, and save the retrieved data. The interactive interface is actually an application that uses the API to provide a high-level interface for ADDS. Free-form query submission is supported for experienced users, and menu-driven query submission is supported for those less experienced with the ADDS and SQL languages. Frequently used queries may be stored in the query catalog, and cataloged queries may be selected and modified by the user before execution.

Queries submitted for execution are compiled and optimized for minimal data transmission cost. Semijoins and common subquery elimination are just two of the query optimization techniques used. A user may submit any number of queries for simultaneous execution. ADDS allows a user to "disconnect" from the execution of a query, which is important for long-running queries. A failed query is automatically restarted, without loss of intermediate results, after the cause of the failure is determined and corrected. Also, query execution may be deferred to nonprime time, thereby decreasing execution costs.

The ADDS system includes geographically distributed mainframes running the VM and MVS operating systems and Sun and Apollo workstations running the UNIX operating system. Therefore, providing a uniform network interface to these systems is important for ADDS development and maintenance. The Network Interface Facility (NIFTY) architecture [Lee et al. 1988] is an extension of the OSI Reference Model [ISO 1982] and provides a uniform and reliable interface to computer systems that use different physical communication networks. An ADDS process on one system can initiate a session with an ADDS process on another system without regard for the multitude of heterogeneous network hardware and software that is used to accomplish the session. Currently, NIFTY supports interprocess communication using SNA, ethernet (TCP/IP), and binary synchronous networks.

ADDS maintains the autonomy of the local database systems and does not require any modifications to local DBMS software. The only communication between ADDS and the local DBMSs is in the form of query submission and data retrieval.

3.1.3 ADDS System Architecture

The layered architecture of the ADDS system is illustrated in Figure 1. Global transactions are application programs composed of one or more global database queries and/or updates. A single query may reference data at several sites. The Global Transaction Interface (GTI) verifies the syntactical correctness of user queries and constructs a global execution plan. The Global Data Manager (GDM) determines the location

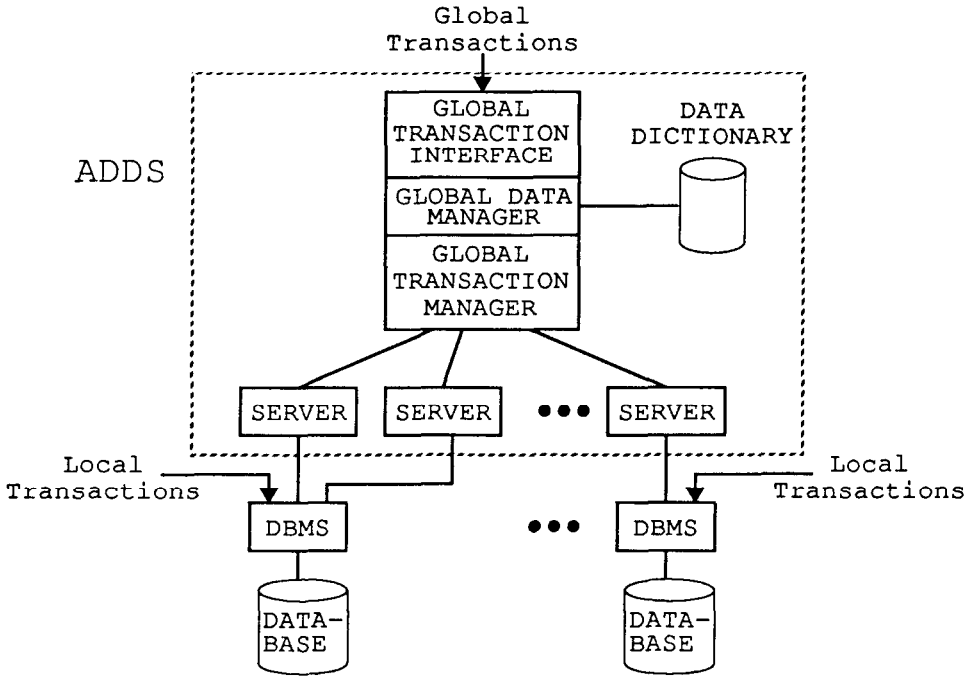

Figure 1. ADDS architecture from [Breitbart et al. 1987].

of the data referenced by a global transaction from the CDB definition in the Data Dictionary. The GDM also manages all intermediate data that is received from the Global Transaction Manager (GTM) during transaction execution. The GTM manages the execution of the global transactions and allocates servers to process global subtransactions.

The GTM uses a two-phase server allocation strategy to guarantee the atomicity of global transactions. All servers remain allocated to a global transaction until the transaction completes. This implies that all local data items referenced by a global transaction remain locked until the transaction completes. Local transactions that have no data items in common with global transactions are unaffected. The GTMs on different network nodes negotiate for server resources when it is necessary for a transaction submitted at one node to access data on other nodes. The servers perform local security verification, then translate the subqueries into the language of the local DBMSs. The servers also perform data conversion as the local data is retrieved and transfer the data to the GTM for further processing.

A "site graph" concurrency control algorithm [Breitbart et al. 1987, 1989a; Thompson 1987] was used in early efforts to support update transactions in ADDS. This general algorithm guarantees the serializable execution of global transactions with permitted local transactions and the absence of global deadlocks [Gligor and Popescu-Zeletin 1985]. In a very active system, however, too many global transactions are aborted. To reduce transaction aborts and increase throughput, ADDS was modified to use a two-phase locking algorithm, with timeouts to guarantee freedom from global deadlocks. A two-phase commit protocol is used to write the results of the global transactions into the local databases. Although no longer used for concurrency control, the site graph is an important tool for guaranteeing global database consistency during commit and recovery

processing [Breitbart et al. 1989b]. Used in this way, the site graph algorithm provides acceptable performance.

3.1.4 ADDS Status and Future Plans

A production version of the ADDS system that supports retrieval transactions has been deployed within Amoco. Prototype support for distributed update transactions, including concurrency control and commit/recovery management, has been integrated into a centralized version of the system; that is, a version in which all global queries are submitted at a single site. Preparations are being made for the limited deployment of the ADDS update system prototype.

Future plans include (1) developing a decentralized version of the transaction management, concurrency control, and commit/recovery algorithms and (2) building tools to simplify the CDB definition process and to maintain synchronization between the CDB definitions and the local database schemata.

Some of the lessons learned during ADDS development and deployment are as follows:

- Introducing distributed data management into a large organization can be a monumental problem.
- A flexible user interface design is necessary to meet diverse and changing user requirements.
- Adequate query optimization and data security are essential for system acceptance.
- Separating the network architecture from the ADDS architecture allowed concentration on important problem areas in both components individually.

3.2 DATAPLEX (General Motors Corporation)

3.2.1 Background on DATAPLEX

Many different kinds of database management systems and file systems are used in the manufacturing industry because of the diverse data management requirements. Historically, there has been no effective means to share these heterogeneous databases. The lack of effective data sharing causes inefficient engineering and manufacturing activities and business operations. Duplicated data at different locations often results in data inconsistency.

A heterogeneous distributed database system is an effective means of sharing data in an organization with diverse data systems. DATAPLEX is a heterogeneous distributed database management system being developed by General Motors Corporation [Chung 1990]. Sections 3.2.2 and 3.2.3 describe the functions and methodologies of DATAPLEX in its target full-function form. Section 3.2.4 describes its current implementation status.

3.2.2 DATAPLEX System Characteristics

DATAPLEX allows queries and transactions to retrieve and update distributed data managed by diverse data systems such that the location of data is transparent to requestors. In this environment, different data management systems can run on different operating systems that may be connected by different communication protocols.

The relational model of data is used as the global data model. Since different data models used by unlike database systems structure data differently, the data definition for each sharable database in the heterogeneous distributed database system is transformed to an equivalent relational data definition or conceptual schema. The conceptual schema is implemented as a set of overlapping relational schemata, one for each location. The relations at each location represent data objects that need to be accessed by users at that location. Consequently, conceptual schemata are neither centralized nor replicated. Thus, in the terminology of [Sheth and Larson 1990], DATAPLEX is a tightly coupled federated system supporting multiple federated schemata.

Use of a common data model eases the problem of providing a uniform user interface. Among several relational query languages, SQL was chosen as the uniform user interface because SQL is widely used

and an ANSI standard has been developed for it. Both interactive SQL queries and embedded SQL programs are supported.

3.2.3 DATAPLEX System Architecture

The above strategies establish the architecture of DATAPLEX. Figure 2 shows DATAPLEX and other elements in a heterogeneous distributed database system. The functions of DATAPLEX are performed by 14 major modules, described here.

The *Controller* module schedules the invocations of the rest of the modules and handles inputs and outputs of the modules.

The *User Interface* and *Application Interface* modules provide interfaces for queries to be entered into DATAPLEX. The User Interface appears to users as a command prompter or a form-oriented query facility. The Application Interface is linked to a compiled application before executing the application.

The *Distributed Database Protocol* (DDBP) module provides communications between the DATAPLEX software at user locations and data locations. Different communication protocols can be used by adapting the DDBP to them.

The *SQL Parser* module checks syntactic errors of SQL statements. The *Distributed Query Decomposer* and *Distributed Query Optimizer* modules prepare distributed queries for execution, with the aid of the Data Dictionary Manager module. The *Translator* and *Local DBMS Interface* modules provide interfaces to the local database systems for execution of local subqueries, and the *Relational Operation Processor* of the user-location DATAPLEX merges the results from the local sites to provide the final query result.

The *Data Dictionary Manager* finds the location of the data referenced by a query and determines the type of the query. There are three different types of queries: user-location query, remote single-location query, and distributed query. The user-location query and the remote single-location query are special cases of the distributed query.

To process a distributed retrieval query, the Distributed Query Decomposer decomposes the distributed query into a set of local queries and a user-location query that merges the results from other locations. (A local query references data from a single location that may be a remote location). The user-location (source) DATAPLEX sends local SQL queries to data-location (target) DATAPLEXs using the DDBP.

The Translator finds query translation information from a translation table that records differences of data names and data structures between the conceptual schema and the local schema. The Translator translates a local SQL query to a query (or program) in a local data manipulation language (DML) using the translation information. The distributed query decomposition method and translation scheme used by DATAPLEX are described in a previous report [Chung 1987]. The Local DBMS Interface sends the translated query to the local DBMS and obtains the local result. The local result is in a report form similar to a relation regardless of the data structure used by a local DBMS.

The Distributed Query Optimizer of the source DATAPLEX schedules an optimal data reduction plan using the statistical information from the target DATAPLEX. The data reduction plan [Chung and Irani 1986] is a sequence of semijoins that consists of local data reduction operations and data moves among computers. Upon completion of the execution of the data reduction plan, the reduced local results are sent to the source DATAPLEX. There the Relational Operation Processor of the source DATAPLEX merges the local results by processing the user-location query.

To process a distributed update query, the Distributed Query Decomposer generates a set of local retrieval queries to identify the specific data to be updated, as well as a set of update queries, one for each relation to be updated.

The *Distributed Transaction Coordinator* enforces two-phase locking on referenced data at the local DBMSs that are involved. Although there are a number of global deadlock detection and avoidance methods

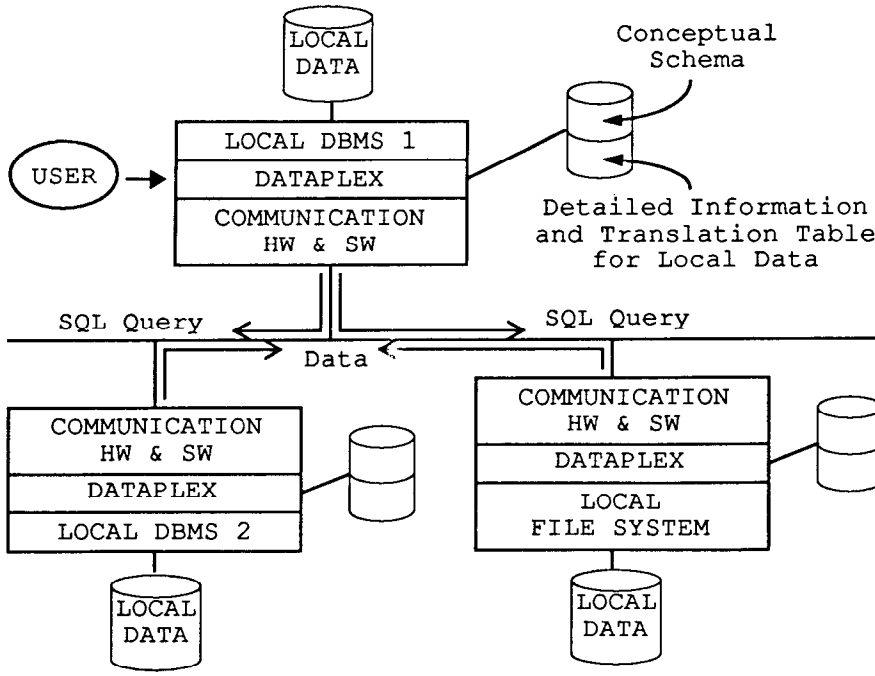

Figure 2. DATAPLEX in a heterogeneous distributed database system.

for homogeneous systems [Elmagarmid 1986], there has not been much research on this topic for heterogeneous systems. Until more research results are available, the time-out method is used initially to handle global deadlocks. After the specific data to be updated is identified by processing the retrieval part, the local update queries are processed, incorporating a two-phase commit to enforce the update atomicity, and then the locks on the referenced data are released. There are also *Security Manager* and *Error Handler* modules.

All modules of DATAPLEX are independent of the local data system except for the Translator and Local DBMS modules. Thus, any data system can be interfaced to DATAPLEX by developing these two modules for them. This architecture is modular and is an open architecture with which functionality and performance can be gradually increased.

3.2.4 DATAPLEX Status and Future Plans

A prototype DATAPLEX was jointly developed in 1986 with a DBMS vendor to show the feasibility of the concepts underlying the DATAPLEX approach. The prototype system interfaces an IMS hierarchical DBMS running under the MVS operating system and an INGRES relational DBMS running on a VAX computer under the VMS operating system.

The features the prototype system provides to users are as follows:

- SQL queries to IMS
- Distributed SQL queries to IMS and INGRES
- 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. In addition, the prototype system checks whether a user is authorized to access IMS data at a segment level using the user id.

Since rapid prototyping was required to show the feasibility of the concept before developing a full-function DATAPLEX,

updates of IMS data and full query optimization were not implemented in the prototype system. In addition to these restrictions, the system supports only a subset of SQL defined to have the following syntax:

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

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

A testbed has been established at General Motors Research Laboratories, with a test distributed database and test transactions. 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 system executes the requests.

Production IMS data have been 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 MMIS database contains a few hundred thousand records. This is about 1000 times bigger than that of the test IMS database. The results of tests using the production database show the prototype system incurs some overhead compared with access using PL/I programs. As the database size grows, the fraction of the overhead to the total processing time decreases. It was observed that the SQL-to-DL/I Translator and IMS Interface modules were the bottleneck in accessing IMS data through the prototype DATAPLEX.

Based on the success of the prototype system, General Motors Corporation has initiated the development of a full-function DATAPLEX system, interfacing IMS, DB2, and INGRES with outside vendors. The full-function system uses full SQL, and the initial version supports distributed retrieval and single-location update. Currently, most of the initial version has been implemented, and completed components are being tested using production applications and data. Subsequent versions will provide the capabilities of distributed update, multiple copy synchronization, and support for horizontal and vertical partitioning.

3.3 IMDAS (National Institute of Standards and Technology, U. Florida)

3.3.1 IMDAS Background: Sharing Data in a Manufacturing Complex

In modern manufacturing systems, two developments are paramount:

- *Industrial Automation*—computer systems controlling and monitoring the physical processes
- *Computer Integrated Manufacturing* (CIM)—direct data sharing among production control systems and the engineering and administrative systems that support them

In most industrial facilities, control, engineering, and administrative systems operate on computer systems and database systems from different manufacturers. They contain independently designed, overlapping databases, with logical and physical differences in the representation of the same real-world objects. These existing systems represent a major investment and support real production. It is not feasible to replace or significantly redesign them.

The *Integrated Manufacturing Data Administration System* (IMDAS) [Barkmeyer et al. 1986; Krishnamurthy et al. 1987; Su et al. 1986] was developed to support a prototype CIM environment–the NBS Automated Manufacturing Research Facility (AMRF) [Nanzetta 1984], a testbed for small-batch manufacturing automation and in-process measurement. The objective was to provide access from many systems to the many sources of manufacturing data, cooperating with existing applications on existing databases while enabling new and modified application programs to access data as needed, insulated from accidental distinctions in location, representation, and access mechanisms.

3.3.2 IMDAS System Characteristics

In the terminology of [Sheth and Larson 1990], IMDAS is a tightly coupled federated system with a single global schema. The integrating data model is the Semantic Association Model (SAM*) [Su 1985], a semantic network data model capable of representing the complex structures and relationships and many integrity constraints found in a manufacturing enterprise. A *fragmentation schema* maps the global model to the underlying databases, supporting both horizontal and vertical partitioning of a given object class.

Existing database systems are front ended by IMDAS modules supporting an internal query interchange form, which is an extended algebra on generalized relations corresponding to the modeled object classes, and a corresponding data interchange form, expressed in Abstract Syntax Notation 1 [ISO 1987a, 1987b]. This common interface is readily mapped onto underlying relational and navigational databases. A library of routines supporting it minimizes the effort involved in integrating new data systems and databases.

The user program phrases queries in an SQL-like language adapted to the model. The query is passed to the IMDAS in string form, rather than precompiled, to permit access by controllers programmed in nonstandard languages. This mechanism can support an interactive interface, although none has been built yet. IMDAS supports both distributed updates (transaction management) and distributed retrievals (query management). The fragmentation schema does not currently support replication, however, which is a significant limitation of the system.

3.3.3 IMDAS System Architecture

The architecture of IMDAS is shown in Figure 3. The lowest level of architecture comprises the data repositories—databases, files, controller memories—managed by commercial DBMSs, file systems, homegrown application-specific servers, and so on. These are the existing data systems on which the IMDAS depends. Each computer system in the enterprise has a Basic Data Server (BDAS), which provides the interface between the local repository managers and the integrated data system. It contains the front-end processes that provide the standard interfaces for the local DBMSs. The BDASs and the DBMSs are the elements that *execute* the data manipulations.

The Distributed Data Servers (DDASs) perform the query processing and transaction management functions. Each DDAS provides the query interface to all application programs within a cluster of computer systems that are its segment of the enterprise and logically integrates the collection of data repositories managed by the BDASs in that cluster into a corresponding segment of the global database. The DDASs *manage* the data manipulations.

The Master Data Server (MDAS) is needed when there is more than one DDAS. It integrates the separately managed segments into the global database and manages transactions that cross segment (i.e., DDAS) boundaries. The MDAS does not "manage" the fully distributed system; it is rather a utility used by the distributed servers to resolve the global model and provide concurrency control for transactions that involve multiple DDASs.

The IMDAS modular architecture permits several "distributed data system architectures" to be built from the same components. A system with exactly one DDAS and one BDAS is essentially a centralized system, whereas a system with one DDAS and multiple BDASs is a distributed system with centralized control. A system with multiple DDASs is a distributed system with distributed control.

An application program issues a transaction to the IMDAS in string form in the data manipulation language. The DDAS query processor in that cluster converts the transaction, expanding application-specific views into standard operations on conceptual generalized relations. If the resulting query can be executed entirely within the DDAS segment, it is passed to the DDAS transaction manager. Otherwise, it is sent to the MDAS. In either case, the query processor reports final status to

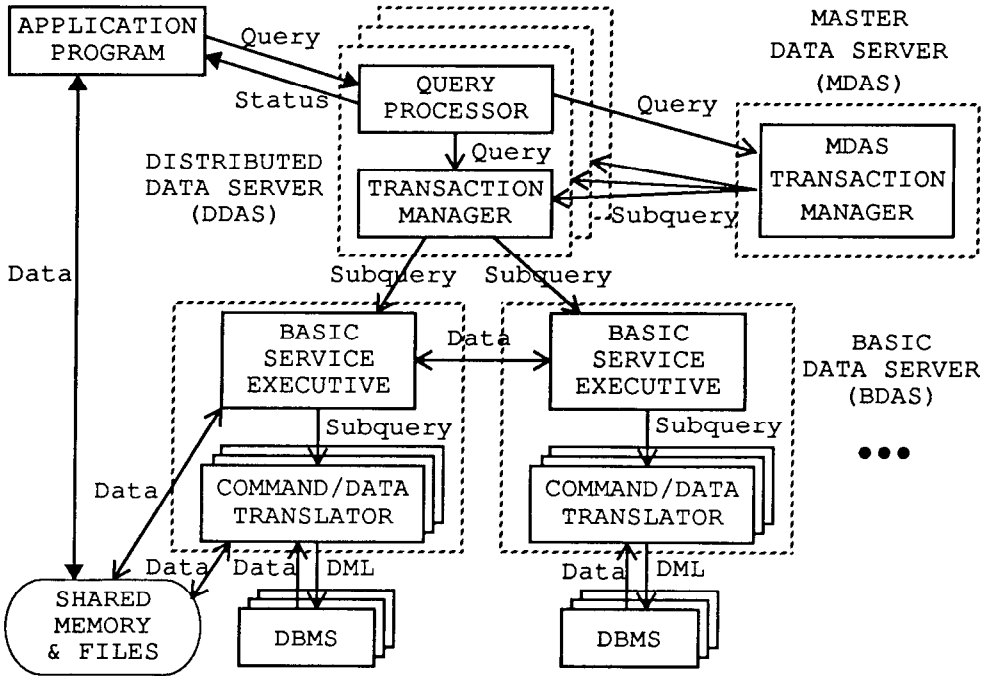

Figure 3. The IMDAS hierarchy.

the user program when the transaction is completed.

The DDAS transaction manager, using a fragmentation schema describing the distribution of its segment of the global model, maps the query into a set of subqueries, each of which operates on elements of the global database managed by an individual DBMS. The mapping algorithm takes into account the capabilities of the target DBMS. Operations that exceed the capabilities of the repository DBMS are routed to a sufficiently capable DBMS, with temporary generalized relations specified to hold the data to be operated on. This is also the mechanism by which information units from multiple DBMSs are integrated.

The subqueries are then dispatched to the affected BDASs, using optimistic commitment in the case of update, because many of the DBMSs have no commitment features at all. That is, individual subtransactions commit independently on the assumption that all will commit. The transaction manager uses a locking mechanism to ensure that simultaneous read/write or write/write access to the same underlying "database" is avoided. In this context a "database" is a modeled collection of information, which is a (possibly proper) subset of the data managed by a single DBMS. Transactions that "conflict" at this level are serialized by the controlling transaction manager, thus effecting distributed concurrency control.

An affected BDAS receives subqueries from the DDAS in the interchange form, specifying the operations to be performed on the local data repositories and the sources and destinations of the associated data. The BDAS converts the subquery to the form appropriate to the designated DBMS and passes it to that DBMS, converting any data involved between the DBMS form and the interchange data form, and reports completion to the DDAS. The BDAS itself accesses any referenced local data that are not managed by a DBMS—files or shared memory [Libes 1985; Mitchell and Barkmeyer 1984]—

converting between the user-specified representation and the IMDAS interchange form. The BDAS also accesses required remote data by communication with the remote BDAS, moving the data directly from producer to consumer without regard to the control path. (For example, if DDAS-1 specifies that data be sent from BDAS-1 to BDAS-2, the data would go directly from BDAS-1 to BDAS-2 without going through DDAS-1.)

The MDAS is essentially a DDAS transaction manager with a fragmentation schema that describes the distribution of the global model over the DDASs, instead of the DBMSs. It accepts transactions from and reports status to the individual DDAS query processors. It sends subqueries to and receives status reports from the individual DDAS transaction managers. Since the MDAS is a clone of the DDAS transaction manager, it can be instantiated in any station that has a DDAS and thus can readily be replaced in the event of failure.

3.3.4 IMDAS Status and Future Plans

IMDAS modules currently exist for VAX computers running under the VMS operating system and for Sun Workstations (which run under the UNIX operating system) using TCP/IP networks. IMDAS interfaces currently exist for RTI/Ingres [Ingres 1986] and BCS/RIM [Boeing Computer Services 1985] systems, for the object-oriented GBASE system [Le Noan 1988], for the AMRF Geometry Modeling System [Tu and Hopp 1987], and for several file systems and shared-memory systems. The current IMDAS is just over 100,000 lines of C and Pascal code, and it represents 15–20 staff years of effort.

A front end for DBMSs using SQL and IMDAS modules for the IBM PC are in development. Modifications to IMDAS to use OSI networks for both internal and external communication and the draft Remote Data Access protocol [ISO 1989] for the user-IMDAS interface are also underway.

Experience in mapping the IMDAS semantic model to different DBMSs and application databases indicates that the use of a semantic integrating model was a wise choice, but the SAM* model itself is not sufficiently flexible. Consideration is being given to replacement of SAM* with a more complete semantic network data model into which many common information models can be translated; for example, SDM, IDEFIX, NIAM, Express, and OSAM*. This implies reworking of the query language, the internal query interpretation, and the mapping onto application databases, each of which has its own strengths and weaknesses, and all of which are appropriate joint research efforts for the 1990s.

On the other hand, the separation of paths for control and data flow at the user interface and within the IMDAS, a clear departure from conventional wisdom, has proven itself to be both effective and efficient and yielded a number of other capabilities, coding elegancies, and design dividends.

3.4 Ingres (Ingres Corporation[1])

3.4.1 Background on Ingres

Ingres Corporation grew out of the INGRES project at the University of California at Berkeley, a research project on relational database technology that began in the early 1970s [Stonebraker 1986]. The company was incorporated as Relational Technology, Inc., in 1980 and changed its name to Ingres Corporation in 1989. The first commercial Ingres database management systems were delivered to customers in 1981. Ingres products currently are available for a wide range of mainframes, minicomputers, workstations, and personal computers under a wide range of operating systems.

Ingres/NET, which provides remote access from an Ingres application at one site to an Ingres database at another site, was first introduced in 1983. Ingres/STAR, which provides transparent access to distributed data, was first introduced in late 1986.

3.4.2 Ingres/STAR System Characteristics

The Ingres DBMS provides access to an Ingres database, which is a named collection of tables. Ingres front-end programs

submit SQL queries to the Ingres DBMS to obtain data stored in the database.

An Ingres Gateway provides a method whereby data stored in other (i.e., non-Ingres) data managers is made to appear as if it were stored in an Ingres database and thus is made available to Ingres front-end programs.

The Ingres/STAR system allows users to access a *distributed database*, which is defined as a collection of tables from one or more Ingres databases. Any set of tables from any set of Ingres databases can be combined to form a new, distributed Ingres/STAR database. This includes not only databases under an Ingres DBMS but also databases accessible via an Ingres/Gateway and, in the near future, other Ingres/STAR databases. A single Ingres/STAR server may service multiple distributed databases, and multiple Ingres/STAR servers may exist in the network. Thus, in the terminology of [Sheth and Larson 1990], Ingres/STAR is a tightly coupled federated system supporting multiple federated schemata.

Access to the Ingres/STAR distributed databases is *transparent* in the sense that once the database has been created, the users of the database no longer need to know anything about the existence of the individual Ingres databases that make up the distributed database. Their contents are now available transparently via Ingres/STAR. Ingres/STAR appears to front-end programs just as if it were a centralized Ingres DBMS. Front-end programs function in the same manner regardless of whether the database being accessed is distributed or not, except for the restriction (in the current release) that within a single transaction only inserts/deletes/updates to data at a single site are allowed. That is, a distributed commit protocol has not yet been implemented in the current release of Ingres/STAR.

Ingres/STAR itself does not deal directly with the physical storage and retrieval of data. Instead it relies upon the Ingres/DBMS and/or Ingres/Gateway components to do this. The Ingres/STAR component communicates with these Ingres data managers (either Ingres DBMSs or Gateways) in the same manner a front-end program would. The same information is communicated. Ingres/STAR sends a query language representation of the desired work, and the data manager replies with the requested data. Figure 4 illustrates a typical configuration of users, data managers, and Ingres/STAR servers.

3.4.3 *Ingres/STAR System Architecture*

As noted above, the Ingres/STAR system builds a distributed database from a number of underlying component databases. In order to provide long-term storage for information about this federation, Ingres/STAR uses a local database called the Coordinator Database (CDB). The CDB holds information on each distributed database concerning

- Which databases are used in the distributed database
- The location of these databases
- The data manager associated with each database
- What tables from each database are included in the distributed database
- Naming (aliasing) information about these tables

Every Ingres DBMS, whether in a centralized or in a distributed system, uses the Internal Ingres Database Database (IIDBDB) to determine the information (sites, disks, users, etc.) necessary to access local or distributed databases. Each site contains one IIDBDB, any number of databases, and some number of servers. Information about each Ingres/STAR distributed database would appear in the IIDBDBs at sites at which queries to the distributed database originate. An Ingres/STAR server must have access to the IIDBDB containing information about each component database of each distributed database that it manages.

Figure 4 depicts a conceptual picture of the various components and databases of an Ingres/STAR system. The functions of Ingres/STAR are provided by seven major modules, described below.

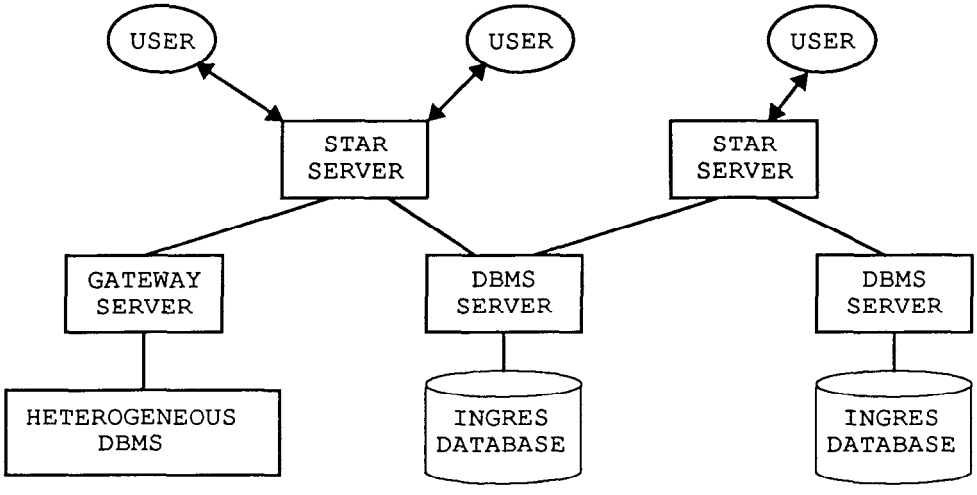

Figure 4. Configuration of Ingres/STAR system components and databases.

The *General Communication Facility* (GCF) provides the intercommunication among instances of Ingres/STAR, Ingres DBMSs, and Ingres/Gateways.

The *Transaction Processing Facility* (TPF), a feature still under development, will be responsible for maintaining Ingres/STAR's transaction system. It will monitor the transaction states of the various Ingres/DBMS, Ingres/Gateway, and Ingres/STAR partners and keep track of the state of distributed transactions. Thus, TPF will know which partners need which instructions during the *prepare*, *commit*, and *abort* portions of a two-phase commit transaction. In the event of an Ingres/STAR crash, it will be TPF's responsibility to resolve any outstanding transactions when Ingres/STAR is running again. TPF will maintain its own log for recovery of distributed transactions. This log will be separate from any log maintained by the DBMSs for recovery of transactions at individual databases.

The *Query Evaluation Facility* (QEF) manages the actual execution of queries. It sends subqueries to the other participants in a session, manipulates the returned results as required, and returns the final results to the Ingres/STAR client.

The *Remote Query Facility* (RQF) receives instructions from either QEF or TPF, formats the instructions, sends them to other participants in the session (Ingres/STARs, Ingres/DBMSs, or Ingres/Gateways), and returns answers to the requestor.

The *Relation Description Facility* (RDF) provides efficient access to catalog information by retrieving it, caching it, and managing the cache.

The *Parser Facility* (PSF) parses the query and passes it on to the *Optimizer Facility* (OPF) in parsed form. OPF plans the method of performing the query. This process is more complex than in an Ingres DBMS because it must take into account the capabilities of the various data managers involved in executing the query (since some may be gateways), the amount of data that must be moved from one site to another, the network speed(s), and the query-processing facilities available at the site of the Ingres/STAR server itself.

3.4.4 Ingres/STAR Status and Future Plans

Gateways are currently available on a production basis for RMS files and RDB databases on VAX computers under the VMS operating system and for DB2 databases on IBM (or compatible) mainframes under the MVS operating system. Gateways are soon

expected to be available on a production basis for SQL/DS databases and IMS databases.

A near-term future release will provide support for a two-phase commit protocol, thereby implementing the distributed transaction management capabilities and allowing distributed updates with full distributed data consistency and recovery. Support is also planned for horizontal and vertical partitioning of tables and for replicated tables.

One of the biggest challenges in designing gateways has been in understanding the capabilities required of the local data managers for participation in distributed operations. A subset of SQL that is supported by all gateways has been designed, and this common SQL is used in the Ingres/STAR product. This common subset is expected to evolve over time.

Similarly, providing a unified view of the system catalogs and data types across a variety of data managers has been a difficult job. A set of standard catalog information has been defined that is available from all gateways. This catalog information is used by Ingres/STAR to obtain information about the local databases.

3.5 Mermaid (Data Integration, Inc.)

3.5.1 Background on Mermaid

Development of the Mermaid® system began at System Development Corporation (now a part of Unisys) in 1982 [Templeton et al. 1987a, 1987b]. The motivation for the project was the requirement in the Department of Defense (DoD) for accessing and integrating data stored in autonomous databases. DoD cannot standardize on a single type of hardware or DBMS and therefore must develop the capability to operate in a permanently heterogeneous environment. After the completion of Mermaid it became clear that this requirement was not unique to DoD. In fact, any large organization may have multiple autonomous databases. In 1989, the development team left Unisys to start Data Integration, Inc., which is continuing development of Mermaid as a commercial product.

® Mermaid is a trademark of Unisys Corporation.

3.5.2 Mermaid System Characteristics

In the terminology of [Sheth and Larson 1990], Mermaid is a tightly coupled federated system supporting multiple federated schemata. In a sense, Mermaid is not a database management system but rather a front-end system that locates and integrates data that are maintained by local DBMSs. Parts of the local databases may be shared with global users.

There are two parts to presenting a single database view to the user of the federated system. First, a federated *view* or *schema* of all or parts of the component databases must be defined. Second, at run time the system needs to translate from the federated schema into the form in which the data are actually stored.

The user is able to use a single query language, SQL, to access and integrate the data from the different databases. The system automatically locates the data, opens connections to the backend DBMSs, issues queries to the DBMSs in the appropriate query language, and integrates the data from multiple sources. Integration may require translation of the data into a standard data type, translation of the units, combination or division of fields, union of horizontal fragments, join of vertical fragments, and/or encoding values.

Several levels of heterogeneity are supported:

- Hardware
- Operating system of the DBMS host
- Network connection to the DBMS host
- DBMS type and query language
- Data model—relational, network, sequential file
- Database schema

Presently the system permits retrieval across databases and updates to a single database. A read transaction may see an inconsistent state of the database, since

local updates may occur in the local databases during query execution. Mermaid minimizes the window of inconsistency by making snapshots of all relations as a first step in processing. The snapshot also reduces relations using the query's selects and projects and translates the schema into the federated schema. Updates to replicated and fragmented relations may cross databases, but the updates to all databases are not necessarily made concurrently. Processing is done interactively, although an application program interface is being developed.

3.5.3 Mermaid System Architecture

As shown in Figure 5, Mermaid has four components: the User Interface, the server, the Data Dictionary/Directory (DD/D) and the DBMS Interfaces. Most of the Mermaid software resides in a server that exists on the same network as the user workstations and DBMSs. The DBMSs may reside on the workstations or on mainframe computers. The system is designed for flexibility and modularity. All components may reside on a single computer, or each may reside on a different computer. In a large system with many users there may be several copies of the components.

At least one Mermaid server must exist to provide a platform for the code. The server contains the optimizer that plans query processing and the controller that configures the system and controls execution. The server runs on a Sun Workstation, taking advantage of its support for many network protocols.

The User Interface includes code to authenticate users, initialize the system, edit queries, maintain a query library, get help with the system, view reports, and optionally manipulate the output returned through other programs. Support is provided for both workstations and dumb terminals and for both single window and multiple window interfaces.

The current Mermaid code provides several levels of access control that make it at least as secure as a commercial DBMS. The first level of access control is a list of users with permission to execute Mermaid from the workstation where the user is logged in. Next, the DD/D is opened and permissions are checked for the specific federated database or view. These permissions must reflect the permissions granted by the administrators for the underlying databases. Mermaid then logs into the local computers and opens the underlying databases.

The query optimizer does dynamic planning. It first locates all required relations and selects one or more copies of replicated relations and some or all fragments of relations. Each local relation is reduced with selects, projects, and joins to other relations at the same site. The size of each intermediate result is returned and used to plan the next step. The optimizer considers network speeds and relative processor speeds when determining the best way to process the query. If there are large relations to be joined, it may perform semijoins between sites before moving data to an assembly site. As much processing as possible is done in parallel.

Each Mermaid component uses the RPC remote procedure call protocol above Transmission Control Protocol/Internet Protocol (TCP/IP) to communicate with the other components, with the possible exception of code residing on the DBMS host. This allows the communication to be the same whether the processes are local or remote. All messages between DBMS Interfaces go through the server, which provides protocol conversion. For example, if one DBMS site uses LU6.2 and another uses TCP/IP, the server would receive a message from the DBMS interface (DBMSI) at the first site using an LU6.2 protocol and send it to the DBMSI at the second site using the TCP/IP protocol.

The Data Dictionary/Directory is a relational database, stored in a commercial DBMS, that contains information about the databases and environment. Figure 5 shows the case of the DD/D residing on the server. The DD/D may also reside on a different computer.

The DBMSs generally reside on different computers. Mermaid has an open architecture that will support the development of interfaces to many types of DBMS. The only requirement is that the data are

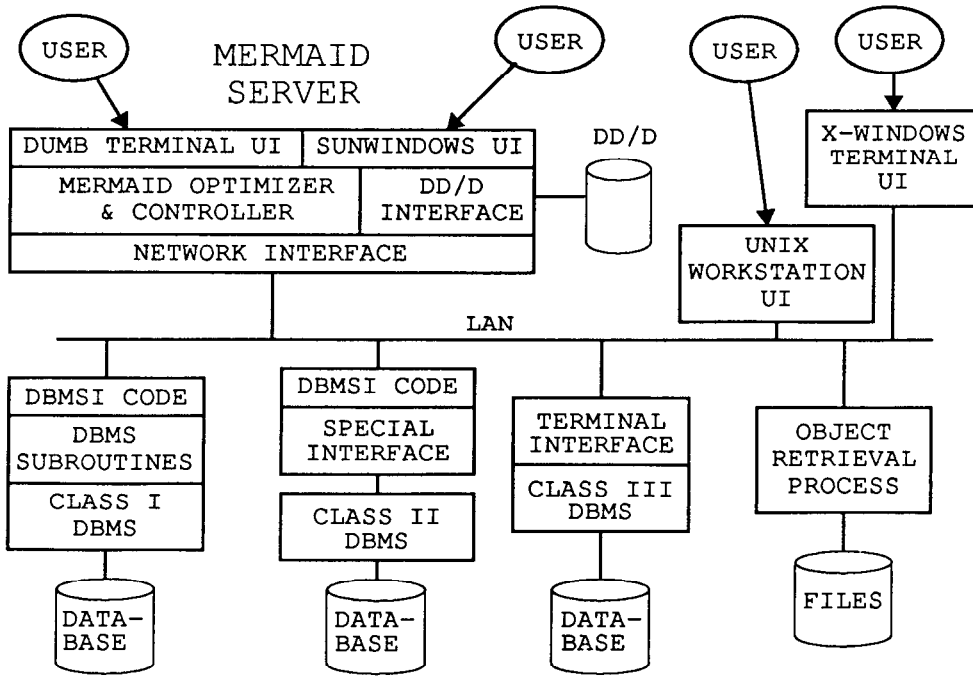

Figure 5. Mermaid system architecture.

managed by a DBMS that provides separation of the application from the data, retrieval of data elements by name (not file offsets), selection of records, an interactive query language, and concurrency control.

There are three generic types of DBMS interface. A Class I DBMS is the favored type. The current INGRES, ORACLE, and IDM interfaces fall into this class. The DBMSI code can be put on the same computer as the DBMS and interfaced to the DBMS through a subroutine call. This provides the most efficient operation and the best error handling. A Class II DBMS either does not support standard network protocols or does not allow the Mermaid DBMS interface code to reside on the same computer as the DBMS. An example of this would be a database machine. The schema and language translators then reside on the server or on another front-end computer. A Class III DBMS is accessed using a terminal emulator interface such as a 3270 emulator from a front-end processor. An example of this would be a DBMS that only supports an interactive terminal interface. This type of interface can be difficult to develop and is weak in error handling.

Mermaid also has the capability to retrieve data from files. A file is a typed object with a retrieval method and a display method. The user selects a set of files of interest by searching on structured fields and listing structured fields and files in the target list. The report looks like a standard report from a relational system except that files are given a symbolic name. The user enters the symbolic name of one or more files he or she wants to see. The method (process) to retrieve the file is started on the computer where the file is stored, and the method (process) to display the file is started in a window on the user's workstation. New file types can be supported by writing the retrieve and display methods.

3.5.4 Mermaid Status and Future Plans

The run-time part of Mermaid was completed at Unisys. Work on the system was started in 1982, and the system was

completely recoded from 1986 to 1987. The total level of effort was 30–40 staff years.

Data Integration, Inc., was founded in April 1989 to continue the product development. The two major missing components in the system at that time were a DBMS-independent DD/D Builder Tool and a system administration utility. These are currently being completed and professional documentation is being written. Additional DBMS interfaces are being developed under contract.

The biggest challenge faced by the developers of Mermaid has been error handling. Mermaid runs above many layers of DBMS, operating system, and network protocol. Each layer has many error conditions that may cause errors in other layers. There is no clear definition of all errors that can occur in each layer or how other layers respond to errors. When an error does occur, it is difficult for the Mermaid code to understand the source of the error and potential cures. It is also difficult to describe some errors to the user.

Another major problem has been coping with new releases of the underlying software. Mermaid frequently encounters problems when new releases of the DBMS, operating system, or network are installed. This poses problems for support of a heterogeneous database system because release control can be difficult in such an environment, and testing of all combinations of underlying software is generally not possible.

3.6 MULTIBASE (Xerox Advanced Information Technology[1])

3.6.1 Background on MULTIBASE

MULTIBASE [Landers and Rosenberg 1982] provides a uniform, integrated interface for retrieving data from preexisting, heterogeneous, distributed databases. It was designed to allow the user to reference data in these databases with one query language over one database description (schema). By presenting a globally integrated view of information, MULTIBASE allows the user to access data in multiple databases quickly and easily. Because there is an integrated schema and a single query language, the user has to be familiar with only one uniform interface instead of numerous local system interfaces.

The MULTIBASE project was initiated in 1980 to develop a software system that would enable organizations to achieve integrated data access without replacing existing databases. The system has three major design objectives. First, it is a general system that is not designed for any specific application area. It can be used without making changes to existing databases and does not interfere with existing application programs. Second, MULTIBASE has been designed to incorporate a wide range of data sources. These data sources encompass the major classes of DBMSs (hierarchical, network, relational) as well as file systems and custom built DBMSs. Third, MULTIBASE has been designed to minimize the cost of adding a new data source. The system is largely description driven, and its modular architecture minimizes the amount of custom software that must be developed for each new DBMS.

3.6.2 MULTIBASE System Characteristics

MULTIBASE uses a data definition and manipulation language called DAPLEX, which is based on the functional data model [Landers et al. 1984; Shipman 1981]. Because the functional model is rich enough to represent relational, hierarchical, and network database schemata directly, the need to translate these schemata and their corresponding operations into a strictly relational model has been eliminated. The system supports ad-hoc query access through several interactive interfaces and an Ada application program interface.

In the terminology of [Sheth and Larson 1990], MULTIBASE is a tightly coupled federated system that provides for the definition of multiple local schemata and multiple federated schemata, or views. Local schemata describe the data available at an individual local DBMS. Views describe integrations of the data described in local

[1] Formerly the Advanced Information Technology Division of Computer Corporation of America.

schemata. Users can query any combination of local schemata or views, and multiple schemata or views can be referenced in a single query. From the user's perspective, views provide complete location transparency. The MULTIBASE view definition language supports horizontal and vertical fragmentation and data mapping of the data contained in the individual local databases. MULTIBASE makes no restrictions on the queries that can be processed over views. The current version of the system does not, however, support updates. Each MULTIBASE location maintains its own directory of local schemata and views.

The MULTIBASE view mechanism is also used to resolve data incompatibilities [Katz et al. 1981] that frequently arise when separately developed and maintained databases are accessed conjointly. Incompatibilities include (a) differences in naming conventions, underlying data structures, representations, or scale, (b) missing data, and (c) conflicting data values. When defining a view, the database administrator applies knowledge of the local databases to determine what incompatibilities might arise and what rules should be used to reconcile them. The rules are included in the view definition, after which they are followed automatically by the system in generating answers to queries.

MULTIBASE performs query optimization at both the global and local levels [Dayal et al. 1981, 1982]. At the global level, the system creates query execution strategies that attempt to minimize the amount of data moved between sites and to maximize the potential for parallel processing that is inherent when multiple distributed databases are accessed. At the local level, the system attempts to minimize the amount of time to retrieve data from a local DBMS by taking full advantage of the local DBMS query language, physical database organization, and fast access paths.

3.6.3 *MULTIBASE System Architecture*

As shown in Figure 6, a MULTIBASE system consists of three major types of components: the Global Data Manager (GDM), one or more Local Database Interfaces (LDIs), and the Internal DBMS.

The GDM is the central component of MULTIBASE, providing query manipulations that require global knowledge. These operations include translation of queries expressed over views into queries expressed over individual local databases, modification of queries to compensate for operations that cannot be performed at the individual local databases, optimization, and generation of plans for the execution of the user's query. All query manipulation in the GDM is performed in an internal form of DAPLEX. The GDM is responsible for management of the global directory, including views, local schemata, authorization information, and descriptions of local DBMS capabilities.

Local Database Interfaces are responsible for receiving queries that have been generated by the GDM, translating these queries from DAPLEX into a form that is acceptable to the local DBMS and passing these queries to the local DBMS to be processed. The LDI receives the data from the local DBMS, translates it into MULTIBASE standard format, and returns it to the GDM. A MULTIBASE system can contain multiple LDIs.

LDIs are only developed for each new local DBMS. The GDM transmits to the LDI the local schemata for the LDI's databases. The local schema contains the information required by the LDI to generate queries against the individual local databases. In general, the GDM does not have to be modified to add a new local database or DBMS. The information about the LDI and local DBMS that is required by the GDM is stored by the GDM in the directory.

The Internal DBMS is a DAPLEX DBMS that is used by the GDM for any processing that cannot be performed at an individual local DBMS. This provides MULTIBASE users with the full power of the DAPLEX query language against any local DBMS. Any operation that cannot be performed by a local DBMS is performed transparently by the Internal DBMS using data that has been retrieved by the LDI under direction of the GDM. The Internal

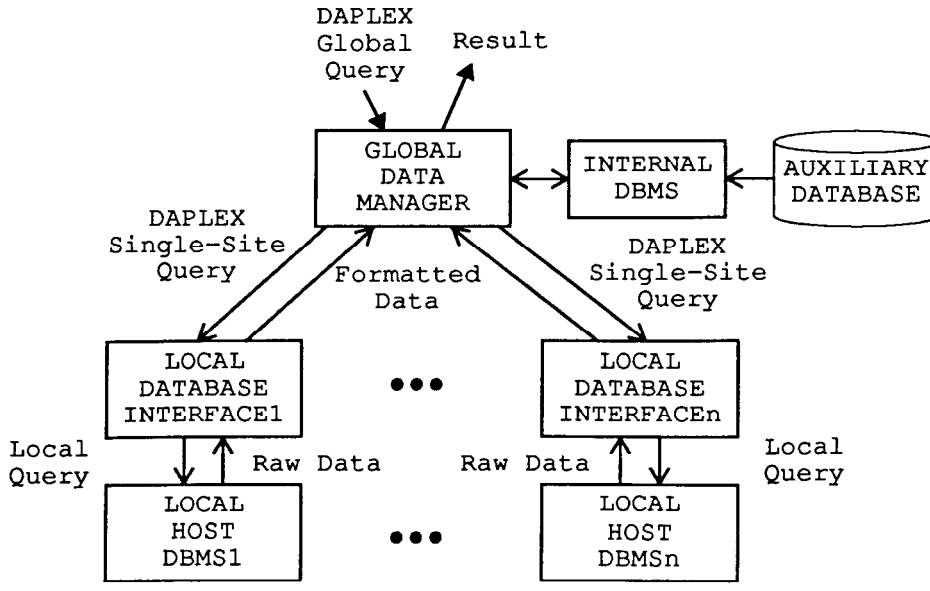

Figure 6. MULTIBASE component architecture.

DBMS can also store auxiliary databases that are used to support resolution of data incompatibilities. For example, auxiliary databases can contain new data that are not available in any local databases, statistics used to determine which data values should be used in case of conflict, and conversion tables that provide a means of performing data transformations that cannot be done using simple formulas.

3.6.4 *MULTIBASE Status and Future Plans*

A MULTIBASE prototype system has been implemented in Ada. It supports the capabilities described above except that only a portion of the global optimization design has been implemented. The GDM software is about 350,000 Ada source lines. It executes on a VAX under the VMS operating system. LDIs have been developed for five DBMSs. They provide access to ORACLE and RIM systems executing under the VMS operating system and to FOCUS, System 2000, and DMR (a hierarchical DBMS developed by the U.S. Army) systems executing under the MVS operating system. The LDIs vary in size from 7000 to 20,000 Ada source lines. To minimize the cost of adding a new DBMS to MULTIBASE, a set of LDI building blocks has been created. These building blocks implement LDI processing that is common to all LDIs and greatly reduce the amount of new software required to create an LDI.

MULTIBASE is currently being used as a component of two systems. One system is a prototype for supporting design, manufacturing, and logistics of large mechanical systems. The other system is a pilot demonstration of the utility of distributed heterogeneous data management for providing integrated access to logistics databases. Both of these efforts require MULTIBASE to interface to existing databases that are not based on the relational model.

Several lessons have been learned during this project that will provide direction for future enhancements to MULTIBASE. Two of the most important of these are the difficulties in handling local system peculiarities and the need for automated tools to support the creation and maintenance of MULTIBASE schemata.

Although MULTIBASE has proved effective at supporting most of the data management capabilities of a wide range of DBMSs, each DBMS has had its own peculiarities (i.e., special data types, operations, or optimization heuristics) that have

turned out to be difficult to support. This has pointed out the need for a more extensible framework, possibly object oriented, that would allow new capabilities to be readily accommodated within the MULTIBASE system.

Experience has indicated that automated tools are needed for administering the dictionary of a system that is integrating databases from many different organizations. Tools are needed to assist both in creating MULTIBASE schemata and in maintaining consistency as changes occur to the local databases.

3.7 SYBASE (Sybase, Inc.)

3.7.1 Background on SYBASE

Sybase, Inc., was founded in 1984 with the goal of bringing a high-performance distributed RDBMS to the market. The initial production versions of the SYBASE SQL Server and the SQL Toolset (application development tools) were shipped in June 1987, and there are currently more than 1000 customers using the product on nearly 30 different hardware platforms. Using a client/server architecture, SYBASE was designed to handle on-line, transaction-oriented applications that require high performance, continuous availability, and data integrity that cannot be circumvented.

The need to integrate a variety of client applications with multiple sources of data is a clear requirement in today's commercial market. Hence, in September 1989 Sybase introduced the Open Server, a product that extends the SYBASE distributed capabilities to heterogeneous data sources. This product complements the Open Client, a client Application Programming Interface (API) used to send SQL or Remote Procedure Calls (RPCs) to an SQL Server. Together they form the Client/Server Interfaces, the basis of the SYBASE approach to heterogeneous distributed databases.

3.7.2 SYBASE System Characteristics

There are two broad categories of distributed databases: on line and decision support. Decision support applications tend to read—but not update—remote data. They are mostly concerned with presenting a decision maker with a unified, single system image of data that is distributed throughout the enterprise. On-line applications, by contrast, involve remote updates and have a strict requirement to maintain local site autonomy. Given the orientation of the SYBASE system to on-line applications, SYBASE is an interoperable system, or a "loosely coupled" federated system in the terminology of Sheth and Larson [1990]. SYBASE attempts to open the architecture as widely as possible to allow any database, application, or service to be integrated into the client/server architecture in a heterogeneous environment. No global data model or schema is enforced. Rather, distributed operations can be supported via application programming or via database-oriented RPCs between SQL Servers. This provides a high degree of site autonomy. At the same time, the SQL Server provides full DBMS support at each location and "prepare to commit" support for a two-phase commit protocol to guarantee recoverability for multisite updates [Gray 1978].

SYBASE is based on the relational model and supports both interactive and programmed access to the SQL Server or any Open Server application. The basic query language is SQL. Multiple SQL statements may, however, be augmented with programming constructs such as conditional logic (if, else, while, etc.), procedure calls and parameters, and local variables. These may be combined into a single database object called a stored procedure. A procedure is an independently protected object and (as in the case of a view) can override the protection of the tables it references. Thus, it is possible to grant execution privileges to a procedure but disallow direct access to the data it references. Procedures can return rows of data and error messages, and they can return values back into programming variables in the application program.

The SQL Server also supports triggers as independent objects in the database [Date 1983]. These have the capabilities

of procedures, with three important extensions:

(1) They cannot be directly executed but rather are executed as a side effect of an SQL delete, insert, or update.
(2) A trigger is an extension of the user's current SQL statement. It can roll back or modify the results of a user's transaction.
(3) Triggers can view the data being changed.

In traditional centralized database systems, users of on-line applications are not given direct update access to a database but rather communicate with an application program that protects the database from the user. This common approach can be called "application-enforced integrity." The legality of any update is determined principally by rules enforced by the application program. Application-enforced integrity is, however, a flawed approach in heterogeneous distributed databases, where the application may be written in a different department or in a different city from the DBA whose database is being updated.

A better alternative in a heterogeneous distributed database is to enforce data integrity within the database itself. Under this alternative, an application at a remote site communicates directly with a database that has sufficient richness of semantics to decide by itself whether the transaction violates any integrity rules. Stored procedures and triggers provide this capability.

The SYBASE Open Server provides a consistent method of receiving SQL requests or remote procedure calls from an application based on the SYBASE SQL Toolset or an application using the SYBASE Open Client Interface and passing them to a non-SYBASE database or application. Whereas Open Client gives users the flexibility to use a variety of front-end packages or applications for accessing and updating data, the SYBASE Open Server allows access to and updating of foreign (non-SYBASE) databases and applications.

At run time an application program issues a database RPC to the distributed database system, which consists of any combination of SQL Servers or Open Servers. If the data are stored in a non-SYBASE source, the Open Server provides the necessary data type and network conversions to allow the Open Client to process the returned data.

SYBASE supports distributed updates that span multiple locations. A two-phase commit protocol, coded in the application, enforces distributed transaction control across multiple SQL Servers.

3.7.3 SYBASE System Architecture

As shown in Figure 7, The SYBASE Open Server consists of two logical components. A server network interface manages the network connection and accepts requests from client programs running Open Client or database RPCs from an SQL Server. Event-driven server utilities in the Server-Library provide the logic to ensure that client requests are passed to the appropriate User Developed Handler and are completed properly. They also ensure that the returned data are correctly formatted for the client program. This enables a SYBASE client application to request processing and exchange information with any application or data management system. The SYBASE Network Interface component of the Open Server appears to the developer and user exactly as an SQL Server interface. It

- Supports multiple connections from multiple clients or SQL Servers
- Supports multiple logical connections on a single network connection to increase network efficiency
- Shields the user from knowledge of underlying networking
- Passes returned data a record at a time in exactly the same format as an SQL Server

The Server-Library enormously simplifies the coding of distributed multiuser server applications. The utilities supplied by SYBASE provide the logic to handle basic server events such as establishing or ending client connections and starting or stopping processes. SYBASE also provides

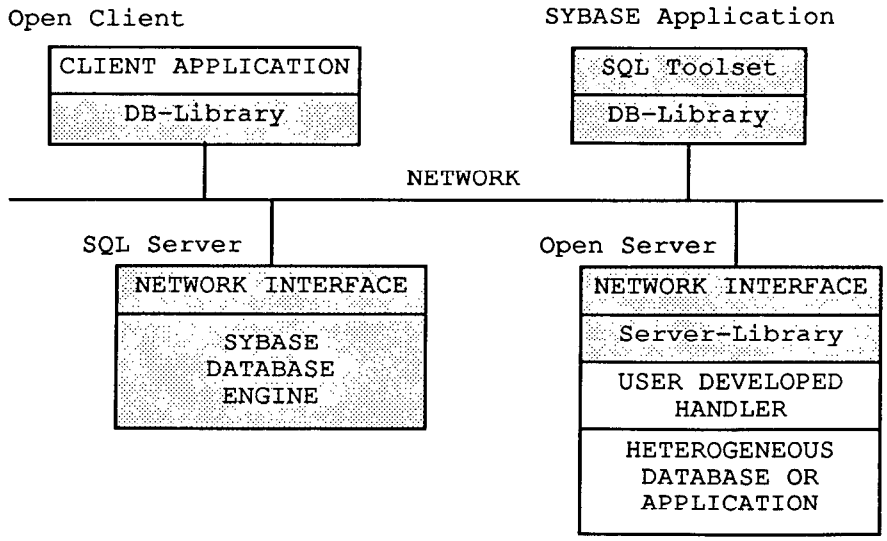

Figure 7. SYBASE open client/server architecture.

utilities to handle SQL and database RPC requests. These utilities are designed to be extended by user-developed handlers. The handlers provide the logic to transmit the RPC or SQL request to the target application or data management system in a predefined predictable way. The user can also define additional utilities to handle events particular to an application or environment. The combination of SYBASE-supplied and user-developed utilities provide the flexibility to develop the appropriate server environment for any application or data source. Through Server-Library the developer can

- Manage the task queue
- Pass requests and parameters to foreign applications or data management systems through the user handler
- Process responses from user handlers—normal, error, and so on
- Collect returned data or parameters from the user handler
- Convert returned data to format compatible with client application

3.7.4 SYBASE Status and Future Plans

The SYBASE Open Server and Release 4.0 of the SYBASE SQL Server, which provides a server-to-server RPC capability, have been commercially available since September 1989. They are in use at numerous customer sites performing on-line applications.

The total effort involved in the development of these products is hard to measure. It would be fair to say, however, that the development of the SQL Server took more than 30 staff years, and the Open Server and RPC enhancements to the SQL Server added approximately 3 staff years. This includes engineering effort only and does not include quality assurance, documentation, and so on. Future plans for the product include moving the two-phase commit protocol into the SQL Server and extending its capabilities to include heterogeneous systems. Access to distributed SQL Servers will be made transparent by means of synonyms and a distributed data dictionary.

4. SUMMARY

As can be seen from the above system descriptions, significant progress has been made in developing heterogeneous distributed database systems for production use. It is, however, certainly not yet possible to buy a system off the shelf that will link all of the popular data models and database

management systems and provide full support for schema integration, distributed query management, and distributed transaction management. Moreover, although the architectural principles are becoming well understood for building a custom system that provides distributed query management, the effort required actually to implement such a system is high. In addition, implementing heterogeneous distributed transaction management is not yet well understood.

There are commercially available systems that bridge a wide variety of types of computers, operating systems, and networks. Gateways are being developed from these systems to various other database management systems based on different data models. Some of these commercial systems offer distributed query management and some offer distributed transaction management, but none offer both, although that is expected to change in the near future. So far these systems offer only limited schema integration capabilities, without system support for horizontal or vertical fragmentation or replicated data, although this is also expected to change in the near future.

Custom systems have been built that encompass a variety of types of computers, operating systems, networks, database management systems, and data models. Some of these systems have rather sophisticated schema integration and distributed query management capabilities. They are, however, just beginning to develop distributed transaction management.

It is important to remember that this is a very fluid landscape. This paper was written in late 1989, and there will undoubtably be advances between that time and the time it appears in print. There will undoubtably be even more advances in the months to follow. Existing systems will improve their capabilities and new systems and vendors will appear.

ACKNOWLEDGMENTS

The authors wish to express their appreciation to the editors and the anonymous referees for the many helpful suggestions that significantly improved this paper.

REFERENCES

BARKMEYER, E., MITCHELL, M., MIKKILINENI, K., SU, S. Y. W., AND LAM, H. 1986. An architecture for an integrated manufacturing data administration system. NBSIR 863312, National Bureau of Standards, Gaithersburg, Md.

BOEING COMPUTER SERVICES 1985. Boeing RIM User's Manual. Version 7.0, 20492-0502, Boeing Computer Services, Seattle, Wash.

BREITBART, Y. J., AND SILBERSCHATZ, A. 1988. Multidatabase systems with a decentralized concurrency control scheme. *IEEE Comput. Soc. Distrib. Proc. Tech. Comm. Newsletter 10*, 2 (Nov.), 35–41.

BREITBART, Y. J., AND TIEMAN, L. R. 1985. ADDS: Heterogeneous distributed database system. In *Distributed Data Sharing Systems*. F. Schreiber and W. Litwin, Eds. North Holland Publishing Co., The Netherlands, pp. 7–24.

BREITBART, Y. J., OLSON, P. L., AND THOMPSON, G. R. 1986. Database integration in a distributed heterogeneous database system. In *Proceedings of the International Conference on Data Engineering* (Los Angeles, CA, Feb. 5–7). IEEE, Washington, D.C., pp. 301–310.

BREITBART, Y. J., SILBERSCHATZ, A., AND THOMPSON, G. R. 1987. An update mechanism for multidatabase systems. *Data Eng. 10*, 3 (Sept.), 12–18.

BREITBART, Y. J., SILBERSCHATZ, A., AND THOMPSON, G. R. 1989a. Transaction management in a multidatabase environment. *Integration of Information Systems: Bridging Heterogeneous Databases*. A. Gupta, Ed. IEEE Press, New York, pp. 135–143.

BREITBART, Y. J., SILBERSCHATZ, A., AND THOMPSON, G. R. 1989b. Reliable transaction management in a multidatabase system. Submitted for publication.

CERI, S., AND PELAGATTI, G. 1984. *Distributed Databases: Principles and Systems*. McGraw-Hill, New York.

CHEN, A. L. P., BRILL, D., TEMPLETON, M. P., AND YU, C. T. 1989. Distributed query processing in a multiple database system. *IEEE J. Select. Areas Commun. 7*, 3 (Apr.), 390–398.

CHUNG, C. W. 1987. DATAPLEX: A heterogeneous distributed database management system. Research Publication GMR-5973, General Motors Research Laboratories (Sept.).

CHUNG, C. W. 1990. DATAPLEX: An access to heterogeneous distributed databases. *Commun. ACM 33*, 1 (Jan.), 70–80. (With corrigendum in *Commun. ACM 33*, 4 (Apr.), p. 459).

CHUNG, C. W., AND IRANI, K. B. 1986. An optimization of queries in distributed database systems. *J. Parall. Distrib. Comput. 3*, 2 (June), 137–157.

DATE, C. J. 1983. *An Introduction to Database Systems*. Vol. II. Addison-Wesley, Reading, Mass.

DAYAL, U., GOODMAN, N., LANDERS, T., OLSEN, K., SMITH, J. AND YEDWAB, L. 1981. Local query optimization in MULTIBASE: A system for heterogeneous distributed databases. Tech. Rep. CCA-81-11, Computer Corporation of America (Oct.).

DAYAL, U., LANDERS, T., AND YEDWAB, L. 1982. Global optimization techniques in MULTIBASE. Tech. Rep. CCA-82-05, Computer Corporation of America (Oct.).

ELMAGARMID, A. K. 1986. A survey of distributed deadlock detection algorithms. *SIGMOD Rec. 15*, 3 (Sept.), 37–45.

ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov.), 624–633.

GLIGOR, V. D., AND POPESCU-ZELETIN, R. 1985. Concurrency control issues in distributed heterogeneous database management systems. *Distributed Data Sharing Systems*. F. Schreiber and W. Litwin, Eds. North-Holland Publishing Co., The Netherlands, 43–56.

GRAY, J. 1978. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmuller, Eds. Springer-Verlag, New York, pp. 393–481.

INGRES 1986. *Ingres*. Relational Technology, Inc, Alameda, Calif. 94501 (Jan.).

ISO 1982. ISO/TC97: *Draft International Standard ISO/DIS 7489: Information Processing Systems—Open Systems Interconnection—Basic Reference Model*. International Organization for Standardization.

ISO 1987a. *International Standard ISO 8824: Information Processing Systems—Open Systems Interconnection—Specification of Abstract Syntax Notation One (ASN.1)*. International Organization for Standardization.

ISO 1987b. *International Standard ISO 8825: Information Processing Systems—Open Systems Interconnection—Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*. International Organization for Standardization.

ISO 1989. Draft proposed international standard 9579: Information processing systems—Open systems interconnection—Generic remote database access service and protocol. M.A. Corfman, Ed., unpublished document ISO/IEC JTC1 SC21 WG3 N845.

KATZ, R., GOODMAN, N., LANDERS, T., SMITH, J., AND YEDWAB, L. 1981. Database integration and incompatible data handling in MULTIBASE: A system for integrating heterogeneous distributed databases. Tech. Rep. CCA-81-06. Computer Corporation of America (May).

KRISHNAMURTHY, V., SU, S. Y. W., LAM, H., MITCHELL, M., AND BARKMEYER, E. 1987. A distributed database architecture for an integrated manufacturing facility. In *Proceedings of the International Conference on Data and Knowledge Systems for Manufacturing and Engineering* (Oct.), Computer Society Press of the IEEE, pp. 4–13.

LANDERS, T., AND ROSENBERG, R. 1982. An overview of MULTIBASE. In *Distributed Databases*, H.-J. Schneider, Ed. North Holland Publishing Company, The Netherlands, pp. 153–184.

LANDERS, T., FOX, S., RIES, D., AND ROSENBERG, R. 1984. DAPLEX User's Manual. Tech. Rep. CCA-84-01, Computer Corporation of America.

LE NOAN, Y. 1988. Object-oriented programming exploits AI. *Comput. Technol. Rev.* (Apr.).

LEE, W. F., OLSON, P. L., THOMAS, G. F., AND THOMPSON, G. R. 1988. A remote user interface for the ADDS multidatabase system. In *Proceedings of the 2nd Oklahoma Workshop on Applied Computing* (Tulsa, Okla. Mar. 18), The University of Tulsa, pp. 194–204.

LIBES, D. 1985. User-level shared variables. In *Proceedings of the Summer 1985 USENIX Conference* (Portland, Oregon, June), The USENIX Association, Berkeley, Calif.

MITCHELL, M., AND BARKMEYER, E. 1984. Data distribution in the NBS AMRF. In *Proceedings of the IPAD II Conference* (Denver, Colo., April), NASA Conference Publication 2301, 211–227.

NANZETTA, P. 1984. Update: NBS research facility addresses problems in set-ups for small batch manufacturing. *Ind. Eng. 16*, 6 (June), 68–73.

SHETH, A., AND LARSON, J. A. 1990. Federated databases: Architectures and integration. *ACM Comput. Surv. 22*, 4 (Dec.).

SHIPMAN, D. 1981. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst. 6*, 1 (Mar.), 140–173.

STONEBRAKER, M., Ed. 1986. *The INGRES Papers: Anatomy of a Relational Database System*. Addison-Wesley, Reading, Mass.

SU, S. Y. W. 1985. Modeling integrated manufacturing data using SAM*. In *Proceedings of GI Fachtagung*, Karlsruhe (Mar.). (Reprinted as *Datenbank-Systeme fur Buro, Technik und Wissenschaft*. Springer-Verlag, New York.)

SU, S. Y. W., LAM, H., KHATIB, M., KRISHNAMURTHY, V., KUMAR, A., MALIK, S. MITCHELL, M., AND BARKMEYER, E. 1986. The architecture and prototype implementation of an integrated manufacturing database administration system. In *Proceedings of Spring COMPCON*.

TEMPLETON, M., WARD P., AND LUND, E. 1987b. Pragmatics of access control in Mermaid. *Q. Bull. IEEE Comput. Soc. Tech. Committee Data Eng. 10*, 3 (Sept.), 33–38.

TEMPLETON, M., BRILL, D., CHEN, A., DAO, S., LUND, E., MACGREGOR, R., WARD, P. 1987a. Mermaid: A front-end to distributed heterogeneous databases. *Proc. IEEE 75*, 5 (May, Special Issue on Distributed Database Systems), 695–708.

THOMPSON, G. R. 1987. Multidatabase concurrency control. Ph.D. dissertation, Oklahoma State University.

TRAIGER, I. L., GRAY, J. N., GALTIERI, C. A., AND LINDSAY, B. G. 1982. Transactions and consistency in distributed database management systems. *ACM Trans. Database Syst. 7*, 3 (Sept.), 323–342.

TU, J. S., AND HOPP, T. H. 1987. Part geometry data in the AMRF. NBSIR 87-3551, National Bureau of Standards, Gaithersburg, Md.

Received November 1988; final revision accepted June 1990.