The CORDS multidatabase project

by G. K. Attaluri
D. P. Bradshaw
N. Coburn
P.-Å. Larson
P. Martin
A. Silberschatz
J. Slonim
Q. Zhu

In virtually every organization, data are stored in a variety of ways and managed by different database and file systems. Applications requiring data from multiple sources must recognize and deal with the specifics of each data source and must also perform any necessary data integration. The objective of a multidatabase system is to provide application developers and end users with an integrated view of and a uniform interface to all the required data. The view and the interface should be independent of where the data are stored and how the data are managed. CORDS is a research project focused on distributed applications. As part of this project, we are designing and prototyping a multidatabase system. This paper provides an overview of the system architecture and describes the approaches taken in the following areas: management of catalog information, schema integration, global query optimization, (distributed) transaction management, and interactions with component data sources. The prototype system gives application developers a view of a single relational database system. Currently supported component data sources include several relational database systems, a hierarchical database system, and a network database system.

Almost every large organization faces a data integration problem in which applications require access to data stored in a variety of data sources, possibly distributed over multiple platforms. The data sources may be diverse, consisting of, for example, file systems, relational database systems, or nonrelational database systems. Typically, each type of data source has its own interface and protocols for retrieving and updating data.

Applications that require data from multiple data sources become complex, expensive to develop and maintain, and directly dependent on the specific data sources. Consider an application program running on a machine that needs to access data in two different database systems. Furthermore, assume that each database system runs on a different machine and that different communication protocols are required to communicate with the machines. The complexity of the application program depends on the level of support provided for connectivity and data integration.

Most modern database systems provide support for remote clients; that is, an application running on a separate machine can transparently access the database systems. Remote access capability provides connectivity—a necessary prerequisite for distributed applications. However, the application program still has to deal with two different

 0018-8670/95/$3.00 © 1995 IBM

interfaces and two sets of error messages and their codes. On retrieval, it also has to perform the processing needed to combine data from the two databases.

Application development is simplified if the two database systems support a common interface.

An MDBS provides an integrated view of data from multiple, autonomous, heterogeneous, distributed sources.

Examples of such interfaces are the Microsoft Open Database Connectivity (ODBC)[1] suite of functions, the X/Open SQL (Structured Query Language) Call Level Interface (CLI),[2,3] and the IBM Distributed Relational Database Architecture* (DRDA*).[4] The application still recognizes that it is dealing with multiple data sources, but now their interfaces are the same. Integration processing, however, is still the responsibility of the application.

Application development is simplified even further if all details of how to access the two database systems are delegated to a separate system. The term *multidatabase system* (MDBS) describes systems with this capability. The objective is to provide the application with the view that it is dealing with a single data source. If a request requires data from multiple sources, the multidatabase system will determine what data are required from each source, retrieve the data, and perform any integration processing needed.

Large user organizations consistently express a strong need for systems that provide better data connectivity and data integration. We believe that the data connectivity problem is more or less solved: applications are now able to retrieve or update data in several different databases on several different platforms. However, simply being able to "get at" the data is not enough.

CORDS (a name stemming from an early group called "COnsortium for Research on Distributed Systems") is a research project focused on distributed applications. It is a collaborative effort involving IBM and several universities. More information about the project can be found in Reference 5. As part of this project, we have designed and prototyped an MDBS, called the CORDS-MDBS, that provides an integrated, relational view of multiple heterogeneous database systems. Currently, five data sources are supported: three different relational database systems, a network database system, and a hierarchical database system. In this paper, we present an overview of the architecture of the CORDS-MDBS and the current state of the prototype implementation. We describe the approaches taken in managing catalog information, schema integration, global query optimization, distributed transaction management, and interfacing to heterogeneous data sources. We also recommend that a few additional facilities be provided by database systems to ease the integration task.

Technical challenges

The objective of an MDBS is to provide an integrated view of data from multiple, autonomous, heterogeneous, distributed sources. Although an MDBS resembles a "traditional" distributed database system, there are major differences, mainly caused by the autonomy and heterogeneity of the underlying data sources.

Autonomy implies that, to a component data source (CDS), the multidatabase system is just another application with no special privileges. It has no control over, or influence on, how the data are modeled by the CDS, how requests are processed, how transaction management is handled, and so on. Simply put, when developing a multidatabase system, we cannot rely on being able to change a CDS; we have to use whatever interface and capabilities a target CDS provides.

Heterogeneity implies that the CDSs may differ in terms of data models, data representation, capabilities, and interfaces. Commonly used models include flat (indexed) files, hierarchical, network, relational, or object-oriented models. Different data models provide different primitives for structuring data, but many other properties and features are typically associated with a data model. These are, for example, the constraints that can

be expressed and enforced, the data definition language, the data manipulation language, and the application program interfaces (API).

We now briefly summarize some of the main challenges for MDBSs in global query optimization, distributed transaction management, schema integration, security, and catalog management in multidatabase systems.

Query optimization—Global query optimization in a multidatabase system is similar to query optimization in a homogeneous distributed database system. However, there are two crucial differences: handling CDSs with different query processing capabilities and lack of query optimization information at the global level in an MDBS.

The query processing capabilities of CDSs may vary greatly, ranging from object-oriented database systems and relational database systems to legacy database systems and file systems. The global query optimizer must decompose a global query into component queries to be processed at the CDSs. It must also determine how and where to perform any integration processing that is needed. To correctly decompose a global query, the global query optimizer needs to know what operations can be performed by a CDS.

To determine an efficient execution plan, the global query optimizer also needs to estimate the cost of processing a component query at a CDS and the amount of output data. The amount of output data produced by component queries is a decisive factor in finding an efficient plan for integration processing. Because CDSs are autonomous pre-existing systems, the global query optimizer may not be able to obtain the necessary information from them to make accurate estimates.

Transaction management—The function of an MDBS transaction manager is to guarantee the properties of global transactions such as atomicity and isolation. The difficulty stems from the fact that local transactions, unknown to the MDBS, may interfere. To ensure the atomicity property of global transactions, the MDBS must, in general, use an atomic commit protocol.[6] However, an atomic commit protocol is not sufficient to ensure correct global schedules. Local transactions may cause a situation where all local schedules are correct, but the global schedule is not.

Schema integration—The key problems in schema integration are related to semantic heterogeneity, for example, the use and meaning of data by different applications, by different administrators, and by different end users. In general, semantic concepts are not defined in database catalogs or application code; they *may* be defined in supporting documentation. Therefore, automatically detecting semantic differences and compensating for them may be impossible. Another important part of the integration process is the detection of conflicts between CDS schemas in representations of the same objects.

The query processing capabilities of CDSs may vary greatly.

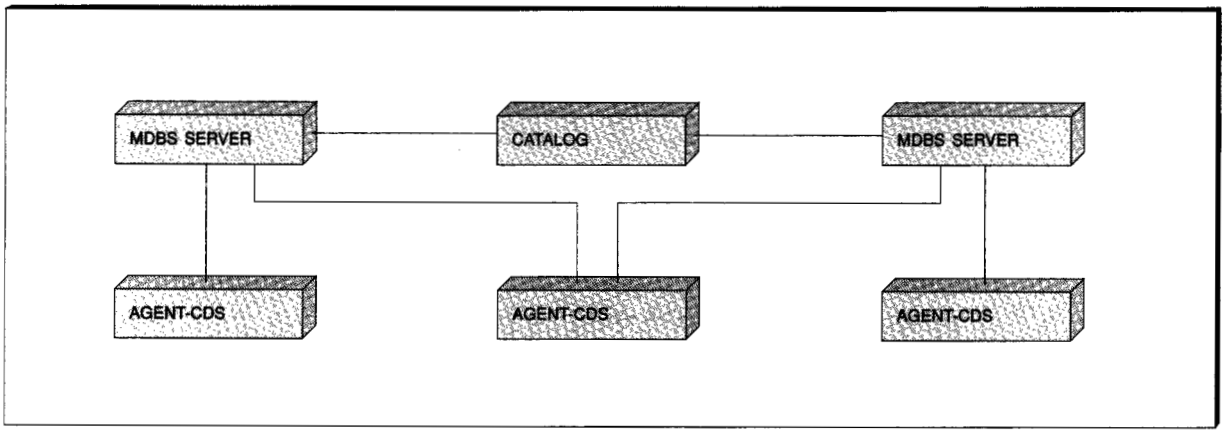
Security—The research community has not addressed the issue of security in MDBSs. Because the CDSs are autonomous, it is unlikely that local security managers will give up or share their control. In addition, heterogeneity means that different CDSs may have different models of security—perhaps even incompatible security systems. The subject of MDBS security is still a major open problem.

Multidatabase catalog—As does any other database system, an MDBS requires a catalog: a data repository storing meta-data and system information. For performance and availability reasons, it must be possible to have multiple MDBS servers running simultaneously at sites distributed in the computing network. At least some of the catalog information must be globally available so that a request can effectively be serviced by any MDBS server.

Architecture and prototype implementation

Within the CORDS project, the MDBS acts as one of the data services offered by the CORDS service environment (CSE) (see Bauer et al.[5,7]). It is de-

Figure 1 A possible MDBS run-time configuration



signed to offer the full functionality of an existing commercial database management system (DBMS).

This section presents the overall architecture of the CORDS-MDBS and briefly describes the current prototype implementation. The prototype system serves as a proof of concept and as a test bed for MDBS functionality. It also helps us to gain some understanding of the practicality of using existing standards and products for modules, interfaces, and protocols.

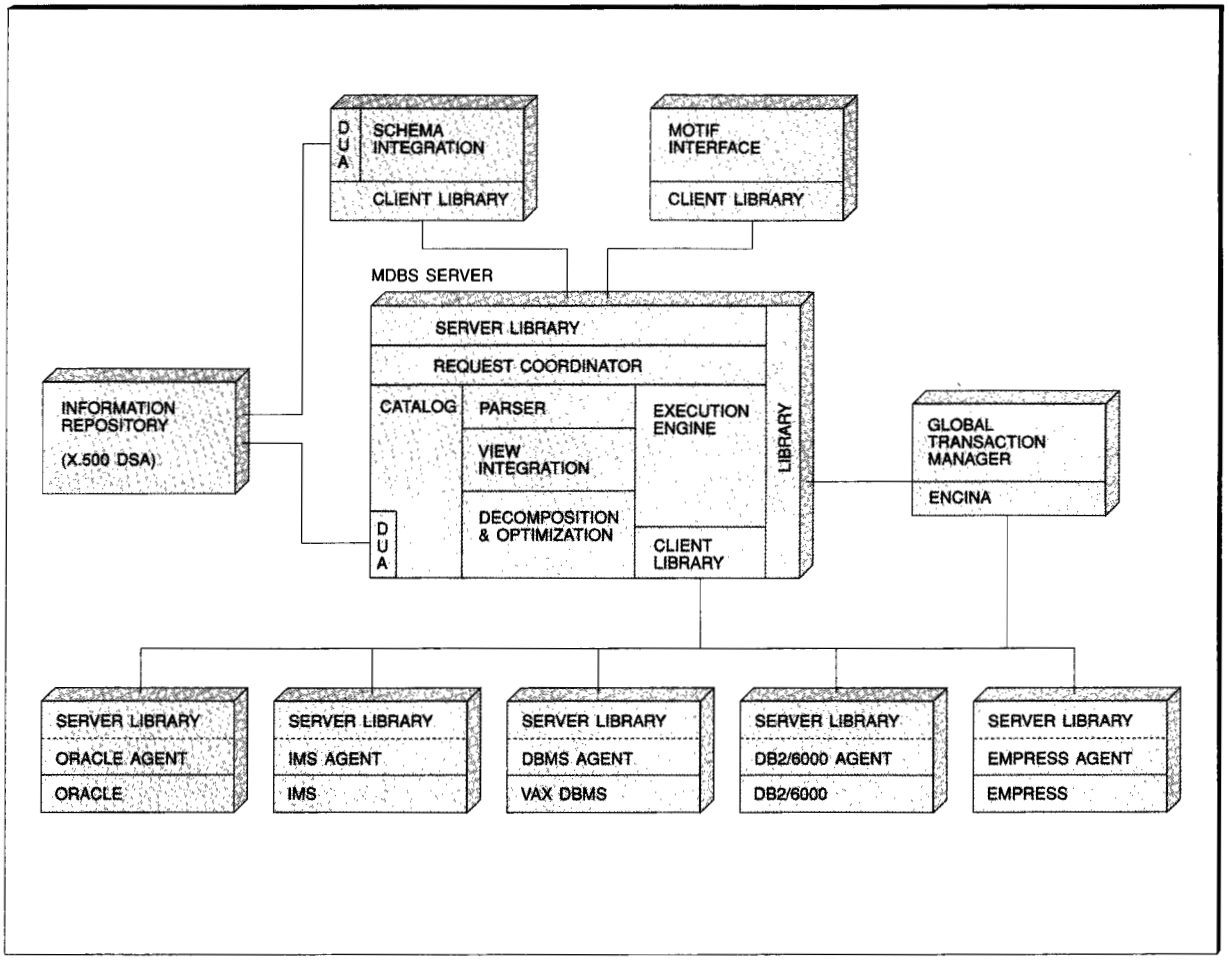
Design objectives. The main design objective was to present applications with a single-image view; that is, from the point of view of an application, the MDBS appears as a single, relational database system. To achieve this view, the MDBS must provide the same functions as a regular database system, and it must hide the heterogeneity and distribution of the underlying CDSs. The design also aims to ensure that the system is scalable and expandable. A practical system needs to scale at least three orders of magnitude: from tens of users on a few platforms to tens of thousands of users on thousands of platforms. In a large system, then, it must be possible to run multiple instances of the MDBS.

Expandability refers to the complexity and cost of incorporating new data sources. The need for easy expandability requires that all CDSs should support the same common data model and present the same interface to the MDBS. If the data model or the interface of a particular underlying CDS, or both, differs from the chosen standard, a CDS-specific MDBS agent can be developed to hide the differences. In our design, MDBS agents are strictly separated from the main MDBS software; no other components of the system are aware of the details of a CDS. There is one exception: an MDBS agent may provide only a subset of the processing capabilities required by the common model. For example, an MDBS agent for an indexed file system may support only single-table SQL queries. It must be possible to describe the capabilities in a generic manner. This information is included in the catalog.

Figure 1 depicts a possible run-time configuration. Six "servers" are distributed at nodes across a communications network: three agent-CDS instances, a catalog server, and two MDBS servers. The term *agent-CDS* refers to an MDBS agent and its associated CDS. The two MDBS servers use the (global) catalog server to maintain MDBS meta-data. They issue component queries to the agent-CDSs and, if required, process the data further to produce the final result.

We believe that the MDBS should present exactly the same interface as MDBS agents. The immediate benefit of this uniformity is flexibility in prototype testing. However, a more important benefit is flexibility in MDBS composition. Since the MDBS presents the same interface as a CDS agent, an MDBS instance can be included as a CDS in any

Figure 2 Main components of CORDS-MDBS prototype



other MDBS. Thus, instead of having a one-level structure of MDBSs, one can build a hierarchy of MDBSs.

Prototype implementation. The main components of the CORDS-MDBS are shown in Figure 2. This structure does not represent our final goal for the architecture of a complete MDBS but it provides a reasonable subset of the functionality of a complete system. The system was built on top of the Advanced Interactive Executive* (AIX*) and Open Software Foundation/Distributed Computing Environment (OSF/DCE**).[8] The amount of prototype code exceeds 200 000 lines.

Communication. Interprocess communication is supported by the MDBS client and server libraries. The client library supports a (draft) version of the Microsoft ODBC interface. It translates ODBC calls into IBM Distributed Data Management (DDM)[9] messages that it ships via Sun RPC (remote procedure call)[10] or the Encina** Transactional RPC[11] according to the DRDA protocol. The server library accepts the RPC calls and translates them back into ODBC calls.

Schema integration. The schema integration component is an environment to support users during the integration process. It consolidates a set of

tools to help with the various tasks involved in the integration process, in particular schema translation, conflict resolution, and schema merging. The current prototype, called the MDBS *View Builder*, supports an X Windows System** interface built on Motif** and provides facilities for schema translation among several data models, for browsing and querying the MDBS catalog, and for creating and managing MDBS views and the transformation functions used for conflict resolution. Schema integration is discussed in more detail in the next section.

Motif interface. The user interface is a simple X-Windows/Motif-based graphical user interface. It is a normal MDBS application, used primarily for testing. A user can edit and submit SQL queries and updates posed against MDBS tables. The interface module submits these queries, via the MDBS client and server libraries, to the MDBS server. The resulting rows are retrieved and presented in a display window.

Request coordinator. As its name indicates, the request coordinator coordinates the actual processing of user requests. A request corresponds to an ODBC call. A single request may require the coordinator to interact with several MDBS modules: parser, view integrator, global query optimizer, execution engine, and transaction manager. The current version of the coordinator can operate in multiuser mode but is single-threaded. In other words, it supports multiple simultaneous connections, but it can only process one RPC call at a time.

Catalog. The MDBS catalog serves a purpose similar to the catalog of a relational DBMS. Catalog data are of two types: (1) *structural data*—descriptions of objects in the system and their relationships, and (2) *statistical data*—mainly statistics from CDSs used during query optimization. The objects described in the catalog include the sites in the network, the CDSs on the sites, the schemas defining the data from the CDSs, and the users. For example, a CDS is described by properties such as the type of the data source, the data model used, if any, and the functionality available; the schemas are presented in relational form, and the information maintained in the catalog includes the names of the tables in the schema, the name and type of each attribute in the tables, and the mappings from the local schema of the CDS to its relational representation.

To achieve scalability, it must be possible to run several instances of the MDBS server distributed in a network. Each of the servers must have access to the same catalog information. This requires a single logical MDBS catalog on top of distributed physical copies. The MDBS catalog must also support a global naming scheme that can be used to uniquely identify objects on distributed heterogeneous CDSs. For these reasons, we decided to implement the prototype MDBS catalog using an X.500 directory.[12]

To achieve scalability, it must be possible to run several instances of the MDBS server distributed in a network.

The catalog module is implemented on top of the EAN X.500 Directory Service,[13] which was extended to provide full transaction support. The catalog module interacts with the directory service via a directory user agent (DUA) that is responsible for querying the directory. The directory service itself is transparently distributed over physically separated entities called directory system agents (DSA). All catalog information is currently stored in the X.500 directory. However, it is not mandatory: the X.500 directory could contain only top-level information and references to other (local) catalogs.

Parser. The parser module is a straightforward SQL parser built using YACC (Yet Another Compiler Compiler). We found that YACC is not an ideal tool for building parsers to be used in long-running, multithreaded servers. Parsers generated by YACC are not reentrant, and they cause memory leaks when syntax errors are encountered. We have subsequently solved both these problems.

View integration. View definitions are expressed in terms of CDS export tables, which define the data available from a CDS in a relational form or in terms of other views. To process a user re-

quest, the request must be transformed into one that is expressed solely in terms of export tables.

The view integration module takes a user query and merges it with the view definitions that are referenced in the query. The module recursively integrates a view definition with a user query so that it can handle a view defined in terms of views. The parse trees representing the views referenced in the user query are stored in the catalog. The output of this module is a single parse tree representation of the merged query parse tree and view definition parse trees. Queries and view definitions may contain simple selects, joins on one site or across multiple sites, unions across sites, and subqueries.

Query decomposition and optimization. All MDBS agents provide the global query optimizer with the same relational interface, even though the underlying CDS may not be relational. The only exception is that an MDBS agent may only support a subset of SQL. The current prototype of the global query optimizer has limited capabilities. It supports SQL SELECT-FROM-WHERE queries with ORDER BY, GROUP BY, and HAVING clauses. It also allows a restricted set of subqueries: a subquery can only refer to tables in a single CDS schema. It uses heuristic rules to perform decomposition and optimization. Global query optimization is further discussed in a later section of this paper.

Execution engine. The execution engine takes a global execution plan, submits the SQL requests contained in the plan to the appropriate CDSs, and then performs integration processing on the resulting data streams. This module is basically the same as any relational execution engine. The current version of our execution engine can perform joins and unions across multiple CDSs. They are performed in a pipelined, data-driven fashion by (internally) buffering intermediate results.

Transaction management subsystem. The transaction manager maintains unique transaction identifiers, guarantees global "serializability," and manages distributed global commitment. The CORDS-MDBS transaction management prototype employs the Encina distributed transaction management toolkit,[11] the X/Open XA interface protocol,[14] OSF/DCE, and the IBM DRDA protocol. The prototype supports both CDSs embedded in a DCE environment and those external to DCE. We first consider CDSs embedded in DCE and then the changes necessary to support CDSs that are not DCE-compliant.

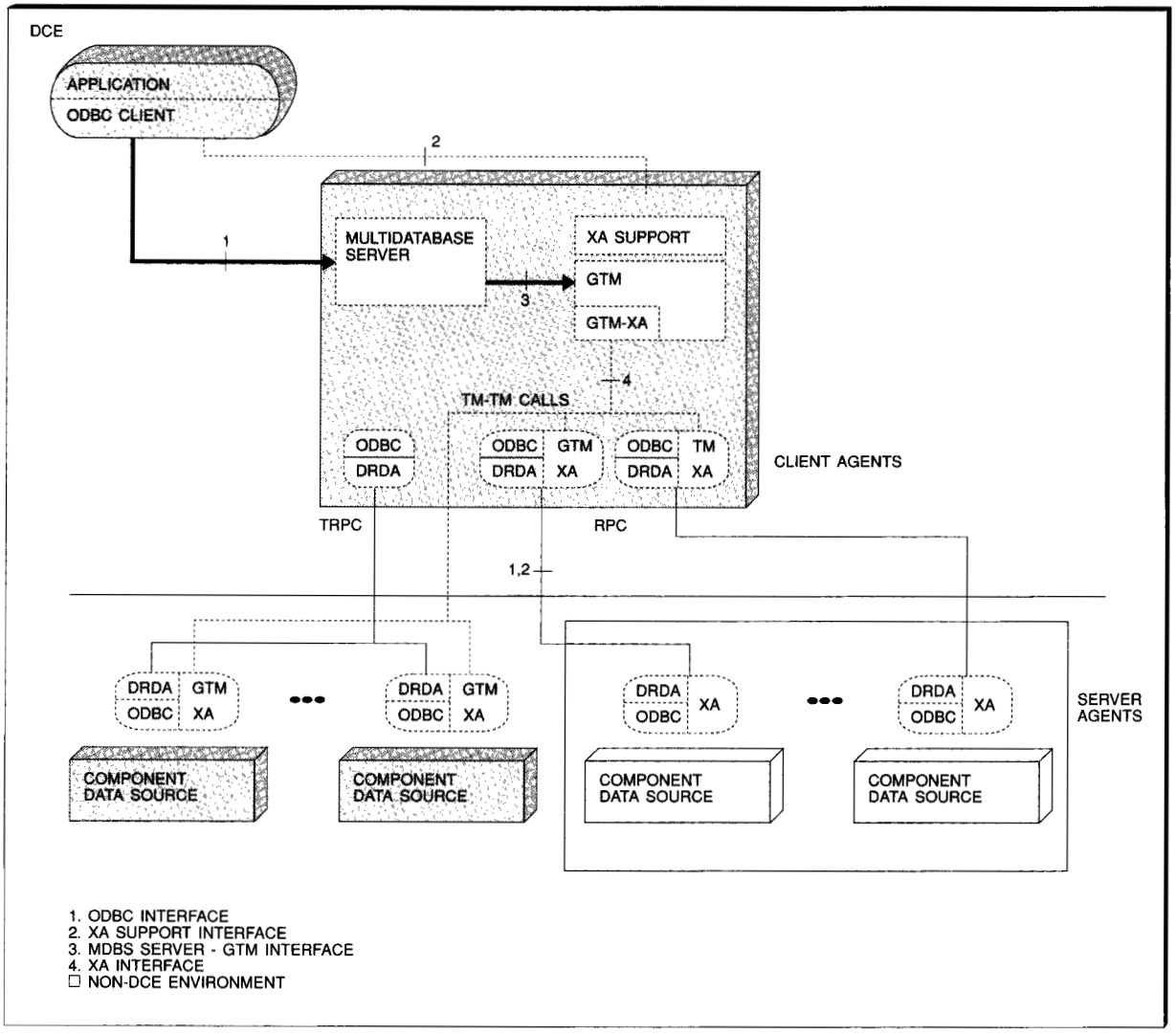
DCE-compliant component data sources. Figure 3 illustrates the CORDS-MDBS transaction management subsystem, its components, and their interfaces. SQL requests are submitted from client applications to the MDBS server using Encina transactional remote procedure calls (TRPC). The MDBS server parses client requests and decomposes them to component requests that are submitted to CDSs. Transactions begin in the client application. The transaction context for a given request is propagated implicitly through TRPCs.

Client agents submit the component requests through TRPCs to ODBC libraries of server agents at the target CDSs. When a server agent receives an ODBC request, it starts a thread that invokes the corresponding native ODBC function from its underlying CDS. With use of the native XA interface, the transaction is then mapped from the thread to a local CDS transaction, in whose context the native ODBC call is executed.

The global transaction manager (GTM) is based on the Encina distributed transaction toolkit and is distributed through linked libraries in each MDBS component: the client application, the MDBS server and client agents, and the server agents. The GTM is responsible for global transaction processing. The transaction context is propagated among distributed components through TRPCs. Each CDS registers itself with the local component of the GTM. The GTM starts and manages local transactions at a CDS through XA calls at the corresponding GTM component and the server agent. Global serializability is guaranteed through a multidatabase concurrency control scheduler that manages the serializable execution and order of commitment of global subtransactions at CDSs.

Transactions are terminated by commit or rollback calls from the client application. Rollback calls may also arise from exceptions during concurrency control scheduling at the MDBS server or CDS, respectively. When an application requests a commit from the GTM, the two-phase commit protocol is initiated at each GTM component participating in the transaction. The MDBS server is selected as the two-phase commit (2PC) coordinator and submits prepare and commit or rollback requests to the GTM components embedded in the server agents.

Figure 3 CORDS-MDBS transaction management subsystem



Non-DCE component data sources. Use of the Encina toolkit in managing distributed transactions mandates that all CDSs run in a DCE environment. When DCE is not available at a CDS site, the server agent is split into two parts: the server agent client running under DCE, and the server agent server running at the CDS site. The server agent client gives the MDBS server the illusion that the CDS runs under DCE by converting all TRPC calls and XA callbacks to (Sun) RPC calls to the native ODBC and XA calls at the server agent server. In effect, the client end of the server agent acts as a protocol converter that converts TRPC and XA protocol calls to simple RPC calls to the server end of the server agent.

MDBS agents. MDBS agents serve two purposes: to provide a standardized interface to a CDS and to simulate required functionality that is absent from, or not exposed by, the CDS.

The CORDS-MDBS prototype currently incorporates five component database systems. Three of them are relational database systems: Oracle**,

DB2/6000*, and EMPRESS**; all three run on AIX. Two are older nonrelational systems: VAX/DBMS** on VMS** (network) and IMS* (Information Management System*) on MVS (Multiple Virtual Storage) (hierarchical). Agents communicate with an execution engine through the MDBS server library which is linked in. Each agent provides an ODBC interface, returns data in a standard format, and conforms to our standard error-handling and reporting rules.

Relational agents—Agents for relational systems implement a subset of the ODBC call suite on top of the native call-level interface of the DBMS. Return codes are mapped, as appropriate, to ODBC error messages.

Nonrelational agents—The agents for IMS and VAX/DBMS are much more complex than the relational agents. Both agents implement an SQL front end on top of the record-at-a-time interface provided by the underlying systems. The current implementation supports SQL SELECT-FROM-WHERE queries, including subqueries, grouping, and aggregation. ORDER BY, set operations, and host variables are not yet supported.

Both nonrelational agents perform query optimization, attempting to exploit the retrieval capabilities of the underlying system. Many queries can be handled by a simple nested-loop algorithm. However, some queries require further processing, for example, unions and sorts. Each agent, therefore, includes a postprocessing engine that handles the processing that cannot be done efficiently by nested loops. The postprocessing engine must be capable of performing all the normal relational operations. Some additional information on these agents can be found in Reference 15.

Schema integration

The schema integration component of an MDBS provides the schema definitions and mappings required to facilitate the global applications of the MDBS. Its main goal is to help identify, and integrate, semantically similar objects in the contributing schemas. This process is made difficult, however, by semantic heterogeneity; that is, semantically equivalent objects are represented with different names, different structures, different types, and different constraints.

The problem of schema integration in an MDBS has received a great deal of attention, and a number of prototype systems that attempt to perform integration have been reported.[16–18] The only commercially available system, to our knowledge, with schema integration facilities is InterViso**.[19] All of these systems only address certain aspects of the integration process. Our work defines a framework for the entire integration process and aims to provide an integrated environment to support schema integration.

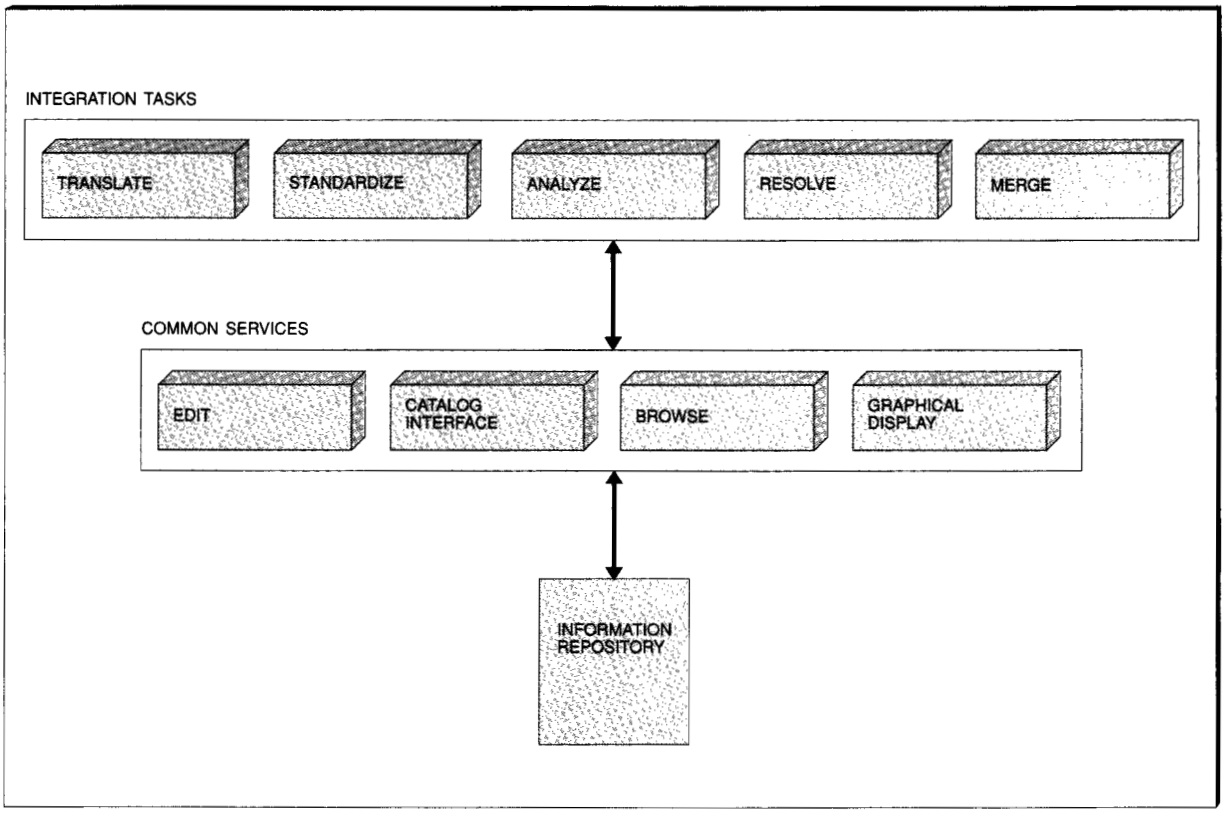
Common data model. The common data model used during integration is an extension of the relational model. Two types of tables are in the model: *export* tables and MDBS *views*. Export tables, as stated previously, present the data available from a CDS in a relational form. MDBS views span multiple heterogeneous databases. They are similar to relational views in that they are not physically materialized but rather are stored as mappings to be invoked whenever an MDBS view is accessed.

MDBS views form the key concept in our approach to integration. They provide a mechanism to define application schemas and to specify the mappings from export schemas to the application schemas. The query defining an MDBS view can be complex. We currently support unions, joins within a single CDS or across CDSs, and subqueries. MDBS views may be defined on top of export tables or other MDBS views.

The common data model also includes *transformation functions*, which are user-defined functions for resolving conflicts. They are specified as part of the MDBS view definition and applied to the attributes of the export tables participating in the view. The functions are executed on data coming from a CDS before the data are presented as part of an MDBS view. Transformation functions may currently be defined in C and then compiled and placed in a function library. A description of a function is stored in the catalog to facilitate searching and retrieval of the functions.

Schema integration environment. The structure for our schema integration environment is shown in Figure 4. The schema integration component, called the MDBS View Builder, consists of a subset of the tools shown. Conceptually, the integration environment consists of three layers: the infor-

Figure 4 Schema integration environment



mation repository, the common services layer, and the integration task layer.

The information repository, as discussed earlier, holds the MDBS catalog that contains the schemas and the mappings between the schemas defined during integration. It links the various tools together by acting as a common information store.

The common services layer contains functions used by all, or at least several, of the tools. It also provides the interface between the integration tools and the information repository. The current functions provided by the common services layer include:

- *Edit*—allows users to modify the definitions of schema objects such as tables, attributes, and mappings during the integration process and to define transformation functions for conflict resolution
- *Browse*—allows users to inspect the contents of the catalog by following relationships between the objects. The current implementation provides a graphical interface that allows users to follow links between catalog entries, to focus on the details of particular entries, and to specify filters that restrict the browsing to particular parts of the catalog.
- *Catalog interface*—the API for the catalog; also used by the edit and browse functions
- *Graphical display*—used by tools to display schemas

The integration task layer consists of a set of tools that correspond to the tasks comprising the schema integration process. These tasks include the following:

- Translate a local schema (or portion of a local schema) into its corresponding representation in the common data model. The current MDBS View Builder provides a number of translators based on structural transformation[20] that translate between relational, entity-relationship, hierarchical, and network schemas. For any pair of data models it is important to identify those schema transformations that are guaranteed to preserve the semantics of the original schema so that information is not lost as a result of the translation.
- Standardize the translated schemas into a normalized form to remove syntactic conflicts such as name conflicts and schema isomorphism conflicts. For example, in two schemas, the names "employee" and "worker" could refer to equivalent objects and would be resolved in an MDBS view definition into a single name. Also, a semantic object may be represented syntactically in several ways; for example, an address may be represented by a single character string attribute "address," or by multiple attributes such as "street," "city," and "province." This would standardize to a single representation by the application of a transformation function such as one to concatenate the values of "street," "city," and "province" into a single string.
- Analyze the contributing schemas and specify the correspondences between objects in the schemas. A schema correspondence describes a set relationship, such as equality, inclusion, or exclusion, between the extensions of related objects in the schemas. For example, an "employee" table in a department database may replicate part of the data in an "employee" table in an overall company database. This would be represented by an inclusion correspondence between the two tables.
- Resolve conflicts detected during the analysis. Objects that were determined to be related may differ with respect to properties such as domain, scale, and precision. Transformation functions are defined to convert data in the extensions of the objects to a common format.
- Merge the processed schemas into an integrated view by defining an MDBS view that contains all of the appropriate transformation functions defined in the previous steps.

The schema integration environment is intended to support users during the integration process in the same way in which CASE (computer-aided software engineering) tools support developers during the software creation process. Users start with the set of local schemas and then perform the various integration tasks. At each stage, the user is supported by specialized tools for the particular task, and by facilities that search the information repository for relevant information and store the results of the stage in the repository for use in later tasks. Details of the prototype system can be found in Reference 21.

Global query optimization

As mentioned previously, multidatabase query optimization differs from traditional distributed query optimization in two respects: handling data sources with different query-processing capabilities and lack of query optimization information at the global level. In this section, we focus on the second issue, that is, lack of information required to accurately estimate the cost and resulting size of a component query to be executed by a CDS. We ignore file systems and consider only CDSs that are database systems.

Little research has been reported on query optimization in multidatabase systems. Lu et al.[22,23] discuss some differences in global query optimization between an MDBS and a traditional distributed database system (DDBS). They also describe a framework for a global query optimizer. Du et al.[24] proposed a calibration method for deriving local cost functions. However, the proposed method has several shortcomings.

Estimating component query cost and output size. An execution plan produced by the CORDS-MDBS query optimizer consists of two parts: component queries and an integration plan. A component query is an SQL query to be executed by an agent-CDS. The integration plan defines how to process the data produced by component queries to produce the final result. (A query may not require any integration processing.) To perform its task, the global query optimizer needs two crucial pieces of information: the estimated cost of executing a component query at a given CDS and the estimated size of the result. However, the statistical data required to compute these estimates may not be available at the global level.

We divide component DBMSs into four classes according to how the cost and output size of a component query can be estimated:

1. Systems that provide estimates of cost and output size. In this case, the global query optimizer does not need to compute the estimates. An example of a DBMS of this type is RDB/VMS.
2. Systems that can explain their execution plan (but the plan does not include cost or size estimates). Many relational systems now include an explain facility that can be used to extract a description of the execution plan for a query. The level of detail varies from system to system, but, typically, the description does not include cost or size estimates. The global query optimizer needs to estimate the cost and output size for each individual component query in the plan. Examples of this type of DBMS are Oracle and DB2/6000.
3. Systems with a catalog (but without an explain facility). We assume that the catalog contains structural information and, at least, some statistical information about the database. Structural information defines the objects in the database: tables, columns, indexes, and so on. Statistical information refers to information such as the number of rows in a table and the number of distinct values for a column. All relational database systems provide at least this level of information. Many older, nonrelational systems do have a catalog (even though it may not be called a catalog) that contains structural information. The global query optimizer needs to predict both the local execution plans and the local cost functions.
4. Systems without a catalog. File systems often do not have an explicit catalog.

For types 2, 3, and 4, the global query optimizer needs to approximate local cost functions. We are investigating three new techniques for estimating the parameters of local cost functions: query sampling, probing queries, and piggybacking.

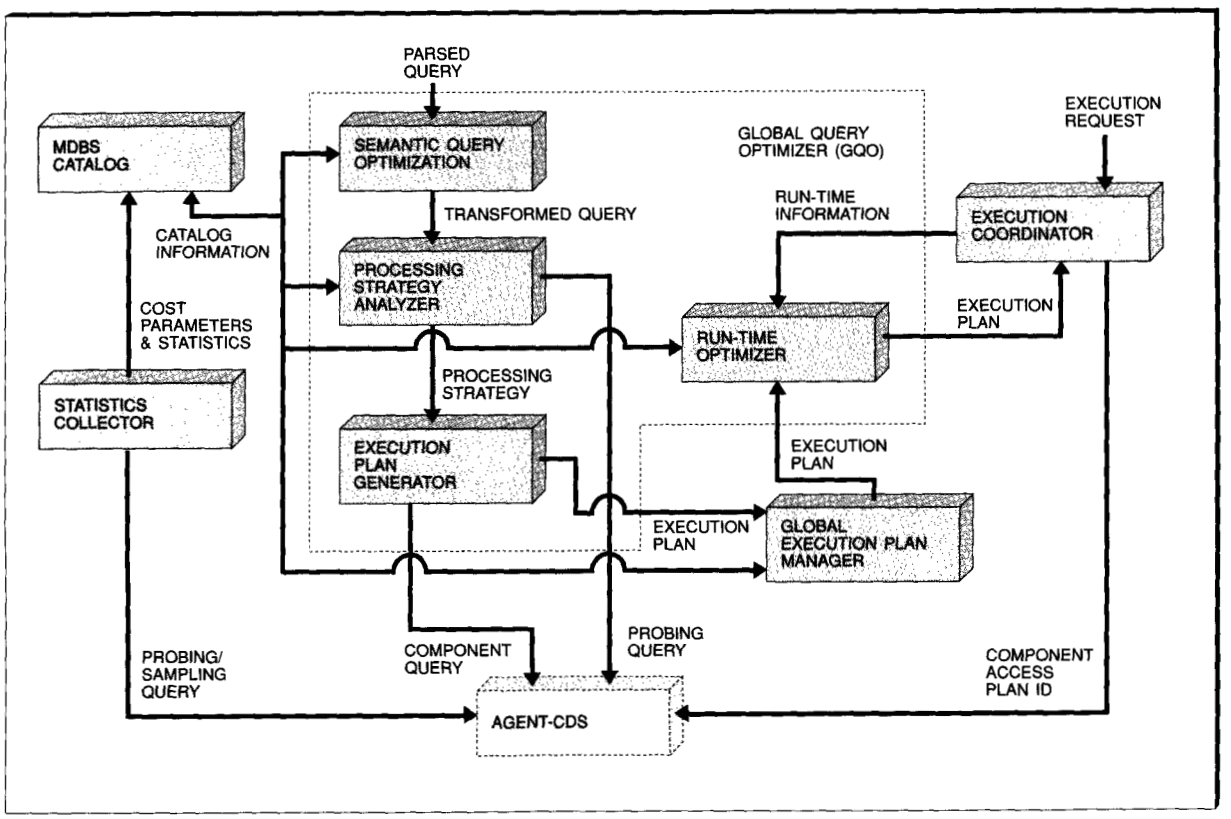
Query sampling. All component queries that can be performed on an agent-CDS are divided into classes such that the execution cost of the queries in each class can be estimated by the same formula. For example, queries that expect to employ the same access method (such as index-based join) can be put into one class. Most likely, they will have similar performance behavior, and therefore, their cost can be estimated by the same formula. Such a classification can be based on the limited information available at the global level, such as the characteristics of queries (unary or join queries, etc.), characteristics of operand tables (indexed columns, etc.), and characteristics of the component DBMS included in the relevant agent-CDS (types of supported access methods, etc). Different agent-CDSs may provide different levels of details of local information. The more information that is available, the better a classification can be obtained.

After classification, a sample of queries is drawn from each class. The sample queries are performed on the agent-CDS, and their costs are observed. Multiple regression is used to establish a cost estimation formula for each query class. During global query optimization, the global query optimizer uses the derived cost formulas to estimate the costs of component queries. The cost estimation formulas can be dynamically revised to reflect the changing environment in an MDBS. More details on this method can be found in Reference 25.

Probing queries. Carefully designed probing queries are issued on an agent-CDS to directly, or indirectly, retrieve some required local information. For example, assume that R_j is known to be a base table in the component database DB_j at site j, but its cardinality is not available in the MDBS catalog. The global query optimizer can then (i) perform a query on the catalog of DB_j to get the size information if access is permitted, or (ii) perform a probing query on R_j, which returns some result from which the cardinality can be estimated. The challenge is to find "cheap" probing queries that allow estimation with high accuracy. More details about this method can be found in Reference 26.

Piggybacking. In addition to exploiting information about intermediate results during query processing, we can also perform additional "side retrievals" on the underlying database to obtain necessary information. We may, for example, include an extra column in the list of output columns in a query and then obtain statistical information about the extra column. Although "side retrievals" are not related to query processing and may slow it down slightly, the information collected can be used to improve the processing of other queries through better cost estimation. In this way, statistical information for the data referred to by popular queries can be maintained in the MDBS catalog. Additional information about this method can be found in Reference 27.

Figure 5 Global query optimization subsystem



Global query optimization subsystem. Figure 5 shows the global query optimizer (GQO) and its connections with other CORDS-MDBS components. The optimizer is not yet fully implemented.

At compile time, when the CORDS-MDBS receives an SQL query from a global user, the parser in the system checks the syntax and semantics of the query using the schema information stored in the MDBS catalog. The semantic query optimization module then rewrites the parsed query using semantic information stored in the MDBS catalog. The objective is to transform the original query to a semantically equivalent query that can be processed more efficiently. For the transformed query, the processing strategy analyzer generates and analyzes alternative strategies for processing the query to select a preferred one. The processing strategy analyzer can perform probing queries on the relevant agent-CDSs to obtain the necessary local information. The processing strategy includes the major decisions about how to transfer data among agent-CDSs, where to execute join operations, and so on. According to the processing strategy, the execution plan generator ships the component queries contained in the plan to the relevant agent-CDSs for local query optimization (done by both MDBS agents and the underlying component DBMSs in the agent-CDSs). A detailed (global) execution plan for the query is then created by the execution plan generator. All execution plans for global queries are managed by the global execution plan manager for later use. An execution plan may be incomplete if some required information is not available at compile time.

At run time, when the CORDS-MDBS receives an execution request for a query, the execution coordinator interacts with the run-time optimizer.

The run-time optimizer retrieves the execution plan from the global execution plan manager. It first checks the validity of the plan. (Some invalid execution plans may, however, only be detected during execution.) If the execution plan is invalid, the global query optimizer produces a new valid execution plan. If necessary, the run-time optimizer performs parametric query optimization to improve or complete the execution plan. It then passes the possibly incomplete global execution plan to the execution coordinator. The execution coordinator coordinates and monitors the data transmissions among execution engines based on the global execution plan. The execution coordinator also returns some run-time information to the run-time optimizer to use when performing adaptive query optimization to improve or complete the execution plan. Improved statistical information is stored in the MDBS catalog and can be used to improve cost parameters. An improved or completed execution plan is then passed to the execution coordinator for further execution.

Multidatabase applications access the MDBS through global transactions.

A utility, called Statistics Collector, is periodically invoked on component databases to collect and update the statistics and cost parameters stored in the MDBS catalog by performing probing and sampling queries. For additional discussion of global query optimization in the CORDS-MDBS, see References 25 through 29.

Managing multidatabase transactions

In a typical DBMS, access to local data is achieved through *transactions*. Transactions result from the execution of a user program segment, written in a high-level language, that interacts with the data manipulation and control facilities of the underlying database system. A transaction manager ensures that the execution of all concurrent transactions is atomic, consistent, isolated, and durable. These four properties are usually called the ACID properties of transactions.

Multidatabase applications access the MDBS through global transactions. A *global transaction* consists of subtransactions that execute as local transactions at the appropriate CDSs. As mentioned earlier, GTM guarantees the ACID properties in all global transactions accessing the MDBS. The GTM must do this in the presence of local transactions, that are not under its control, and with the possibility of failures at CDSs.

We assume that each CDS provides a local transaction manager (LTM), whose function is to guarantee the ACID properties of the *transactions* running at its site. Unfortunately, the heterogeneous nature of the various LTMs and the desire to preserve the local autonomy of the data sources participating in an MDBS make transaction management in an MDBS environment difficult. Multidatabase composition, global serializability, global atomicity, deadlock handling, and accessing nonstructured data all pose important challenges for transaction management in an MDBS.

Our work on transaction management focuses on implementing techniques proposed in the literature for dealing with transaction management issues. In particular, we investigate the empirical performance of these algorithms to determine their applicability in the real world. Furthermore, we attempt to extend the algorithms to deal with the more complex issues of multidatabase composition and integrating transaction management schemes for large objects.

Related work. The work on multidatabase concurrency control focuses on guaranteeing global serializability through rigorous scheduling and forced conflicts.[30] Garcia-Molina et al.[31] completed a comprehensive survey of alternative schemes for guaranteeing global serializability in multidatabase systems. The survey also covers deadlock handling and concurrency control schemes in which either serializability or autonomy constraints are relaxed.

Work on 2PC emulation through the two-phase commit agent method (2PCA) was first published by Wolski and Veijalainen.[32,33] Alternative schemes for guaranteeing global atomic commitment with one-phase commit resource managers are described by Gray[34] and Breitbart et al.[35]

Work on *superdatabases*, published by Pu,[36] is the only known work on multidatabase composition. A superdatabase is analogous to a multidatabase and can have other superdatabases as CDSs. In this work, superdatabase composition is restricted to a strict static tree. Our work on multidatabase composition focuses on dynamic composition.

Research on an interoperable environment consisting of structured and nonstructured data sources is in the initial stages. Several papers have recognized its importance and highlighted some issues. A survey of issues in MDBS transaction management and nonstructured data sources (large data objects and long transactions) can be found in Reference 37. The multigranularity locking scheme[38] is a well-known concurrency control scheme for hierarchically structured large objects. Variations or extensions to this scheme have been described in References 39 and 40.

Multidatabase composition. Current work on MDBSs assumes a two-level architecture comprising a monolithic multidatabase server and a collection of component database systems. This centralized view is restrictive and does not scale to include multiple cooperating multidatabase systems. In the CORDS-MDBS transaction subsystem, we eliminate these restrictions and focus on a design of multiple cooperating peer global transaction managers distributed on a communication network. With this design, a global transaction can span multiple servers, causing some multidatabase servers to act as both multidatabase systems and component database systems. Therefore, multidatabase systems can be composed arbitrarily and dynamically to resolve any global transaction that spans at least two multidatabase environments.

We have shown that guaranteeing multidatabase serializability[41] at multidatabase servers is sufficient for guaranteeing global multidatabase serializability for a dynamic hierarchically composed set of multidatabase servers.[42] This follows from the property that multidatabase serializability guarantees both atomicity and a consistent ordering of global transactions as they execute from leaf component database sites to the root multidatabase server. However, no sufficiency conditions have been developed for global serializability in arbitrary execution lattices.

Nested transactions[43,44] provide useful conceptual tools for propagating work among multidatabase servers and encapsulating failure recovery. Nested transactions also furnish two useful properties: intratransaction parallelism and failure isolation. Unlike flat transactions, nested transaction blocks do not have to execute serially. A nested transaction can be decomposed into smaller independent granules, or subtransactions, that run in parallel. Subtransaction boundaries act as a *firewall* to failures, and unlike flat distributed transactions, a subtransaction failure does not necessarily imply the failure of the whole global transaction. These properties permit multidatabase servers to delegate concurrent transactional work to other multidatabase servers and limit the propagation of transaction failures across servers. Furthermore, nested transactions provide static structural power for composing strict transaction hierarchies while still maintaining global serializability.

More detailed discussions on dynamic multidatabase composition, concurrency control, and recovery in the CORDS-MDBS can be found in References 45 and 46.

Global serializability. Generally, conflict serializability is adopted as the correctness criterion for the execution of concurrent transactions at a data source. We say that a multidatabase schedule is *locally serializable* if all the schedules at component data sources in the MDBS are conflict serializable.[6] A multidatabase schedule is *globally serializable* if it is locally serializable, and the relative serialization order of global subtransactions is the same at each data source.

Ensuring that schedules are globally serializable is achieved by ensuring that global subtransactions execute in the same relative order at each data source. It is easily done when global subtransactions conflict *directly* at the component data sources. Here, the local concurrency control mechanism of the LTM and an atomic commit protocol guarantee synchronous global subtransaction commit orders at the data sources. However, because of local autonomy constraints, local transactions are not under the control of the GTM and can unwittingly cause *indirect* conflicts among global transactions, resulting in nonserializable schedules. Even the serial execution of global transactions does not ensure global serializability. In general, global serializability cannot

be achieved in a multidatabase environment without placing restrictions on the local concurrency control mechanisms.

Global serializability schemes. We now briefly describe some concurrency control schemes that can be used to ensure global serializability given an atomic commitment protocol, such as the *two-phase commit* (2PC) protocol.[6] Note that the assumption of an atomic commitment protocol compromises execution autonomy at component data sources. If all CDSs guarantee locally serializable execution histories, the following concurrency control mechanisms guarantee globally serializable multidatabase transaction executions.

Forced conflicts. One method for guaranteeing global serializability works by forcing direct conflicts among global transactions at component data sources.[30] This method uses a special data item, called a *ticket*, that is maintained at each local site. A single ticket is required for each CDS, but tickets at different component data sources are distinct. Only global transactions are allowed to access a ticket. Moreover, each global transaction executing at a CDS is required to read the ticket value, increment it, and write the incremented value back to the database. Therefore, ticket values maintain the serialization order of global transactions at CDSs. Before a transaction commits, at a CDS, it sends its ticket value to the GTM. The global concurrency control scheduler of the GTM uses ticket values to maintain a global serialization graph of all uncommitted global transactions. The scheduler guarantees serializability by either avoiding cycles for conservative concurrency control schemes, or breaking cycles through transaction abortion in more aggressive schemes.

Strongly recoverable data sources. Given some knowledge of the properties of the various local schedules that are generated by the local concurrency control schedulers, effective global concurrency control schemes can be devised. Of particular interest is the notion of a *strongly recoverable* schedule.[47,48] Strongly recoverable schedulers produce serializable schedules in which the order of transaction execution and serialization orders are the same. If all CDSs generate strongly recoverable schedules, the serial execution of global transactions ensures global serializability. Various concurrency control mechanisms discussed in the literature—such as 2PL, time-stamp ordering,[49] and optimistic methods,[50] for example—can be easily modified to generate strongly recoverable schedules.[48,51]

Rigorous component data sources. Some concurrency control mechanisms generate local schedules that are even more restrictive than strongly recoverable schedules, and can, in turn, lead to even more efficient global scheduling schemes. One such concurrency control scheme is the strong-strict 2PL protocol that generates *rigorous* schedules. Rigorous schedulers are strongly recoverable but, moreover, prevent *read-write*, *write-read*, and *write-write* conflicts among uncommitted transactions. In Reference 47, it is shown that, if local DBMS schedulers are rigorous and the GTM guarantees the atomic commitment of all global transactions, the global schedule is serializable. Other protocols can be easily modified to generate rigorous schedules. For example, basic time-stamp ordering can be made rigorous by blocking transactions that either try to read or write data that were previously written by an uncommitted transaction or try to write data that were read by an uncommitted transaction.[47]

The global atomicity property is ensured by using an atomic commit protocol.

Global atomicity. Global atomicity specifies that either all subtransactions of a global transaction run successfully at their local CDSs, or they all abort. Recovery specifies that the effects of a global transaction are undone completely after it is aborted or remain entirely durable after system failures. In general, the global atomicity property is ensured by using an *atomic commit protocol*.[6] Once CDSs are locally recoverable, global atomic commitment also guarantees global recovery. An atomic commitment protocol requires that each participating CDS provides a *prepared state* for each subtransaction. The subtransaction should remain in the prepared state until the coordinator

decides whether to commit or abort the transaction. However, to preserve the execution autonomy of each of the participating CDSs, it must be assumed that they do not export a prepared state for global subtransactions. In such an environment, a CDS can unilaterally abort a subtransaction any time before it commits. This condition leads not only to global transactions that are not atomic, but also to incorrect global schedules.

In the CORDS-MDBS, the data sources that do not support the 2PC protocol emulate it through the *two-phase commit agent* (2PCA) algorithm.[32,33,52] The 2PCA intercepts all transactions from the GTM and passes them to the local transaction manager at the component site. All commands but the prepare message are forwarded. When it receives a prepare message, the 2PCA determines whether the corresponding transaction is ready to commit or abort, and responds to the coordinator accordingly. If an agent prepares to commit, but the corresponding local subtransaction unilaterally aborts at the component site, the 2PCA recovers by resubmitting the local transaction.

Deadlock detection and handling. Global deadlocks occur whenever there is a cyclic wait for locks among transactions that run at CDSs that use a locking mechanism for local concurrency control. Deadlock avoidance and detection schemes depend on the avoidance or detection of cycles in a lock wait-for graph (WFG). In a lock WFG, each node corresponds to a transaction, and there is a directed edge between two nodes if one corresponding transaction is waiting for a lock from the other. A WFG is a centralized data structure, and its maintenance incurs delays and communications overheads in a decentralized environment. Although this problem is not unique to MDBSs, it is further exacerbated by autonomy constraints that prevent component databases from exporting deadlock control information to the GTM.

We attempted to resolve the deadlock detection problem heuristically by setting time-out intervals on global transactions. If a global transaction holds a lock longer than a specified time-out period, it is aborted. Heuristic detection can be dangerous if time-out intervals either increase aborts or hold resources too long.

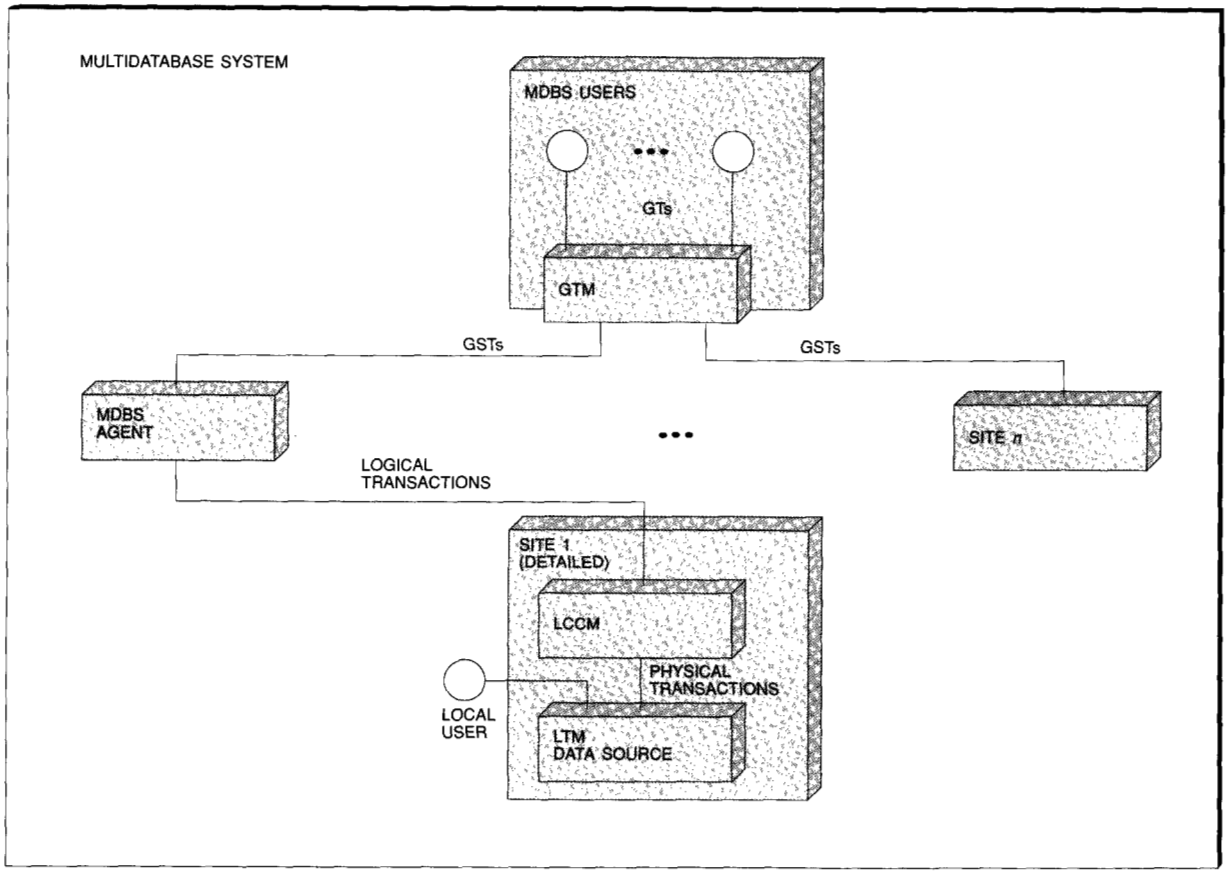
Nonstructured data sources. Computer-aided design (very large-scale integration, mechanical and software engineering), geographical data, and multimedia (voice and image) are emerging as important application areas. These areas differ from the structured ones in their data modeling tools, nature of application programs, and types of storage structures and access methods. Moreover, they require an interoperable environment of structured data sources (for example, relational systems) and nonstructured data sources (for example, object-oriented database systems and file systems). Transactions would be an important part of such an environment. The CORDS-MDBS is intended to provide transactions across structured and nonstructured data sources. This task involves several challenges; the important ones are mentioned below.

First, nonstructured data sources, similar to ones that are structured, employ transactions to support concurrent access and withstand failures. However, objects in nonstructured data sources are normally large and lack a uniform structure, and their application programs model long and complex design processes that involve human interaction. Large and unstructured data objects can cause problems in an MDBS environment. Because of the absence of structure, an entire object has to be locked by a transaction to prevent conflicting transactions from sharing. If entire data objects are locked, the number of lock and other recovery-related operations is minimized, but the concurrency of this and other data sources in the MDBS is degraded. Alternatively, a data object can be granularized into small subobjects so that unused parts of a data object are not held up. Efficient lock management is required to offset the overhead of such fine granularity locking.

Second, global transactions cannot be allowed to lock data objects indefinitely. Because a distributed global transaction can take a relatively long time, the resulting adverse effect on local transactions may not be acceptable to the administrator of the CDS. The GTM must ensure that global subtransactions share data objects with local transactions in a fair manner and with a lower priority.

Finally, because of the autonomy and nonmodifiability of a CDS, any granularization of objects has to be implemented at the MDBS level. The implementation must guarantee the atomicity and serializability of transactions sharing the subobjects it created. It must do so even in the pres-

Figure 6 LCC for nonstructured data sources



ence of transaction abortions and site failures at the data source.

Logical concurrency control in nonstructured data sources. We are working on a *logical concurrency control* (LCC) scheme to manage large, unstructured data objects of nonstructured CDSs efficiently. Whereas the LTM uses a data object as the unit of concurrency control, the LCC scheme enables a data object to be granularized and shared among multiple conflicting transactions concurrently. As a result, LCC could offer a much higher concurrency than the CDS alone. LCC would also facilitate specialized granularization schemes for various types of objects (that is, separate schemes could be employed for two- or three-dimensional objects, for example).

Figure 6 illustrates the use of LCC in MDBS transaction management. The LTM offers the physical transaction service to its users, namely *local users* and the *logical concurrency control manager* (LCCM). The physical transaction service treats a data object as a unit of concurrency control and forbids concurrent access by multiple conflicting transactions. In locking terminology, a data object is the smallest lockable unit. The LCCM is responsible for implementing an LCC scheme and exports the *logical transaction service* to the MDBS agent. The LCCM invokes multiple physical transactions, each dedicated to a data object. Each of these physical transactions fetches its data object from the LTM, granularizes it into smaller subobjects, and offers those subobjects to logical transactions. Thus, in locking

terminology, the LCCM supports a subobject of a data object as a lockable unit.

The presence of LCCM improves the concurrency of logical transactions from the MDBS agent; otherwise, its presence is transparent to the MDBS agent. The MDBS agent can assume that its logical transactions are being executed on the data source directly. The LCCM ensures the atomicity and serializability of logical transactions to support this notion.

Following are the building blocks of an LCCM:

- LTM interface: The LCCM uses this interface to create and manage physical transactions for carrying out tasks described above.
- MDBS agent interface: The MDBS agent uses this interface to have its logical transactions processed by the LCCM.
- Logical transaction–physical transaction mapping: This component maintains the association between the logical transactions of the MDBS agent and the physical transactions of the LCCM. It requires stable storage to record these associations so as to withstand site failures.
- Concurrency control and recovery: This component schedules the operations of the physical transactions of LCCM. It is responsible for ensuring the serializability and atomicity of transactions.
- Logical transaction analysis: The LCCM analyzes logical transactions to recognize an "unused" subobject of a data object locked by a logical transaction. Unused subobjects are allocated to other transactions to improve concurrency.
- Data object granularization: The main contribution of an LCCM is object granularization. This component is application-domain-dependent. It granularizes data objects so that the object structure suits the applications on hand.

The concurrency control and recovery component is the most involved because, for each data object, concurrent operations of logical transactions have to be supported through a single LTM physical transaction.

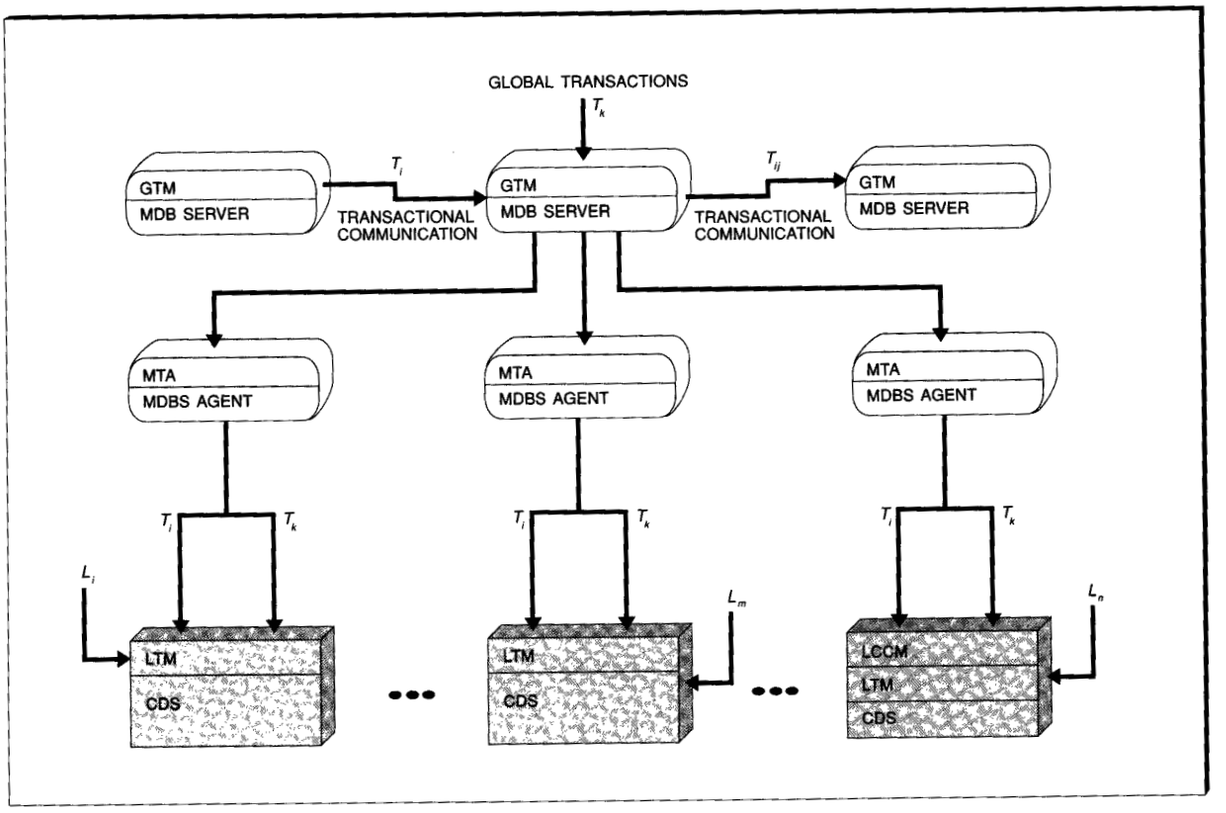
We proposed an LCC scheme in Reference 53. We described the concurrency control and recovery mechanisms and important implementations of an LCC assuming a locking based data source and area-wise granularization of multidimensional data objects. We are implementing LCC schemes on two types of data (text and multidimensional data) using ObjectStore**,[54] an object-oriented database system.

ObjectStore does not employ explicit object or subobject locking but uses two levels of isolation: page locking and object check-in and check-out. A page lock is implicitly acquired by a transaction when it accesses a part of an object located on that page. The lock is released when the transaction terminates (commits or aborts). Page locks guarantee the isolation of transactions. An object is checked out by an application into its area for "long duration" access by its transactions. It is not accessible to other applications until it is checked back in. Of course, if the object is allowed to have multiple versions, new versions can be created and checked out by these applications. If objects are declared as single-versioned, object check-out and page locking are limited forms of object and subobject locking. In our implementation of LCC schemes, check-out and check-in is used for object locking.

Clearly, concurrency control (locking or any other scheme) contributes significant overhead in implementing an LCC scheme. Subobject locking is more involved than locking a table or a page in a relational system for two reasons. First, the lock space is large and consists of all possible subobjects of data objects. Second, determining whether locks on two subobjects conflict is nontrivial because of their nonuniform size. A solution is to model subobjects as one or multidimensional ranges. Ranges can be dynamically indexed to lock or unlock subobjects efficiently. An indexing algorithm for this purpose has been described in Reference 55.

Multidatabase transaction management subsystem. The transaction management subsystem consists of a global transaction manager (GTM), a set of local transaction managers (LTM) at CDSs, and multidatabase transaction processing agents (MTA) that run as part of the MDBS agents at CDSs. The configuration of these components is illustrated in Figure 7. The GTM executes global transactions from both multidatabase servers, T_i, and application programs, T_k. These transactions are decomposed into nested subtransactions and submitted to CDSs through the MTA. Notice that for nonstructured CDSs, the nested subtransactions are submitted to

Figure 7 CORDS-MDBS transaction management subsystem



the LCCM at the CDS. The LCCM creates a logical transaction and relays it to its corresponding LTM.

Transactions at CDSs are committed by the 2PC protocol. If the CDS supports the 2PC protocol, the MTA simply relays the global transaction to the LTM at the CDS site. Otherwise, the MTA acts as a 2PC agent. In the GTM, the global concurrency control scheduler and recovery manager coordinate commit orders at component sites through the MTA. The interaction between the GTM and MTA during global concurrency control depends on the global concurrency control scheme in use. For example, with implicit tickets, 2PC messages are enough; however, in the optimistic ticketing scheme, ticket conflict orders need to be piggybacked on 2PC messages.

Finally, a global transaction, T_i, may contain references to data items at CDSs under the control of another multidatabase server. These references are decomposed into a nested subtransaction, T_{ij}, and shipped to the remote multidatabase server through some form of transactional communication. If the delegated transaction is successful, the nested subtransaction commits. Otherwise, it aborts. The failure isolation property of nested transactions limits the scope of aborts to the remote multidatabase server, making it possible to either resubmit T_{ij} or abort the global transaction, T_i. A more detailed discussion of the CORDS-MDBS, and its relationship to the CORDS distributed transaction services model is given in Reference 56.

Conclusion

Data integration systems may provide many different levels of service. The simplest systems may provide nothing more than connectivity, that is, the ability for an application to access data stored in multiple database systems. The next log-

ical step is to provide a uniform interface, hiding some of the details among the underlying data sources. At the next level of service are systems that support distributed queries; that is, a single SQL query may reference data in multiple databases. Many DBMS vendors have added, or are currently adding, support for distributed queries into their systems. It represents a significant step forward, but additional services are needed. A full-function MDBS should also provide:

- A globally available catalog so that multiple MDBS servers can be run at the same time
- Schema integration tools to ease the task of importing and integrating schemas
- Global transaction management to allow distributed transactions
- Global security services

Integration of database systems would be greatly simplified if the underlying CDSs provided a few additional features. Three highly desirable features are explained below. We offer these as suggestions to standardization bodies and industry consortia in the database area.

The first feature is to define a common interface for retrieving the estimated cost and output size of a query. This feature would greatly simplify MDBS query optimization. It has been claimed that this interface requires agreement on a common cost unit. We believe that such agreement is unnecessary: Each system can provide its cost estimates in whatever units it prefers. The only requirements are that the estimates provided by a system be consistent and have sufficient resolution. The global query optimizer can scale the estimates obtained from different systems, and the appropriate scale factors can be obtained by running calibrating queries.

The second feature is to define a way in which an MDBS can inform a CDS that it has more stringent serialization requirements. To guarantee global serializability, the serialization order of global subtransactions at each site must be consistent, and the global transaction manager must be able to somehow inform a local transaction manager of what serialization orders are acceptable. We see two possible approaches: one implicit and one explicit. In the implicit approach, a global transaction manager would inform a local transaction manager that all of its global subtransactions must have a serialization order consistent with the order in which it submits 2PC prepare requests. The global transaction manager would then process all 2PC prepare requests serially. A drawback of this implicit approach is that the global transaction manager cannot submit prepare requests for the next transaction until all local transaction managers involved in the previous transaction have responded. This drawback can be overcome by allowing the global transaction manager to include an explicit serialization number in 2PC prepare requests.

The third feature is to define a way that an MDBS can generate unique transaction identifiers and have a transaction identifier passed along to all CDSs involved in the transaction. It would enable a CDS to recognize when requests arriving through different routes are in fact part of the same global transaction and should be treated as one transaction. If they are treated as separate transactions and conflict on some data item, the global transaction will deadlock itself.

Acknowledgments

We would like to thank the following participants in the CORDS-MDBS project who have contributed to performing the research investigations or the prototype implementation, or both, reported in this paper: Pravin Baliga (X.500 Catalog Interface), Lauri J. Brown (prototype implementation), Zenith Keeping (EMPRESS agent), Brian Minard (view integration and schema integration), Shahrokh Namvar (EMPRESS agent), Glenn N. Paulley (query optimization, IMS agent, ORACLE agent), Wendy Powley (prototype implementation), Zhanpeng Wang (schema integration), Weipeng Yan (user interface and query optimization), and Jianchun Zhang (schema integration). We would also like to thank Ann Gawman of the IBM Toronto Laboratory for her excellent editorial suggestions.

This research was supported by the Centre for Advanced Studies of the IBM Toronto Laboratory and by the Natural Sciences and Engineering Research Council of Canada.

Cited references

1. *Microsoft ODBC Application Programmer's Guide*, draft edition, Microsoft Corporation, Redmond, WA (1991).
2. *Data Management: SQL Call Level Interface (CLI)*, X/Open Company Limited, CA (1992).
3. *X/Open Portability Guide: Data Management*, 1st edition, X/Open Company Limited, Englewood Cliffs, NJ 07632 (August 1988).
4. *Distributed Relational Database Architecture Reference*, SC26-4651-0, IBM Corporation (1990); available through IBM branch offices.
5. M. A. Bauer, N. Coburn, D. L. Erickson, P. J. Finnigan, J. W. Hong, P.-Å. Larson, J. Pachl, J. Slonim, D. J. Taylor, and T. J. Teorey, "A Distributed System Architecture for a Distributed Application Environment," *IBM Systems Journal* **33**, No. 3, 399–425 (1994).
6. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Series in Computer Science, Addison-Wesley Publishing Co., Reading, MA (1987).
7. M. A. Bauer, N. Coburn, D. L. Erickson, P. J. Finnigan, J. W. Hong, P.-Å. Larson, and J. Slonim, "An Integrated Architecture for Distributed Applications," *Proceedings of the 1993 CAS Conference*, Centre for Advanced Studies, Toronto Laboratory, IBM Canada, Ltd., 844 Don Mills Road, North York, Ontario, Canada M3C 1V7 (1993).
8. *Introduction to OSF DCE*, Open Software Foundation, Cambridge, MA (1991).
9. *IBM Distributed Data Management Level 3 Architecture: Implementation Programmer's Guide*, SC21-9529, IBM Corporation (1990); available through IBM branch offices.
10. J. R. Corbin, *The Art of Distributed Applications*, Sun Technical Reference Library, Springer-Verlag, New York (1991).
11. *Encina: Product Overview*, Transarc Corporation, Pittsburgh, PA (1991).
12. *X.500 Directory Services 1992*, International Telegraph and Telephone Consultative Committee, Geneva (1991).
13. G. Neufeld, B. Brachman, M. Goldberg, and D. Stickings, "The EAN X.500 Directory Service," *Journal of Internetworking Research and Experience* **3**, No. 2, 55–82 (June 1992).
14. *CAE Specification—Distributed Transaction Processing: The XA Specification*, X/Open Company Limited, Reading, Berkshire, United Kingdom (1991).
15. G. N. Paulley, "Engineering an IMS SQL Gateway," *The 1993 Workshop on Interoperability of Database Systems and Database Applications*, Fribourg, Switzerland (September 1993).
16. C. Chung, "DATAPLEX: An Access to Heterogeneous Distributed Databases," *Communications of the ACM* **33**, No. 1, 70–80 (January 1990).
17. U. Dayal and H. Hwang, "View Definition and Generalization for Database Integration in a Multidatabase System," *IEEE Transactions on Software Engineering* **10**, No. 6, 628–644 (November 1984).
18. A. Motro, "Superviews: Virtual Integration of Multiple Databases," *IEEE Transactions on Software Engineering* **13**, No. 7, 785–798 (July 1987).
19. *InterViso User's Manual*, Data Integration Inc., Los Angeles, CA (1992).
20. J. R. Cordy, C. D. Halpern, and E. M. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects," *Computer Languages* **16**, No. 1, 97–107 (January 1991).
21. P. Martin and W. Powley, "Database Integration Using Multidatabase Views," *Proceedings of the 1993 CAS Conference*, Centre for Advanced Studies, Toronto Laboratory, IBM Canada, Ltd., 844 Don Mills Road, North York, Ontario, Canada M3C 1V7 (October 1993), pp. 779–788.
22. H. Lu, B.-C. Ooi, and C.-H. Goh, "On Global Multidatabase Query Optimization," *SIGMOD Record* **21**, No. 4, 6–11 (December 1992).
23. H. Lu and M.-C. Shan, "On Global Query Optimization in Multidatabase Systems," *Second International Workshop on Research Issues on Data Engineering*, Tempe, AZ (1992), p. 217.
24. W. Du, R. Krishnamurthy, and M. C. Shan, "Query Optimization in Heterogeneous DBMS," *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, Vancouver, British Columbia, Canada (1992), pp. 277–291.
25. Q. Zhu and P.-Å. Larson, "A Query Sampling Method of Estimating Local Cost Parameters in a Multidatabase System," *Proceedings of the 10th International Conference on Data Engineering*, Houston, TX (February 1994), pp. 144–153.
26. Q. Zhu, "Query Optimization in Multidatabase Systems," *Proceedings of the 1992 CAS Conference, Volume II*, Centre for Advanced Studies, Toronto Laboratory, IBM Canada, Ltd., 844 Don Mills Road, North York, Ontario, Canada M3C 1V7 (November 1992), pp. 111–127.
27. Q. Zhu, "An Integrated Method of Estimating Selectivities in a Multidatabase System," *Proceedings of the 1993 CAS Conference, Volume II*, Centre for Advanced Studies, Toronto Laboratory, IBM Canada, Ltd., 844 Don Mills Road, North York, Ontario, Canada M3C 1V7 (October 1993), pp. 832–847.
28. Q. Zhu and P.-Å. Larson, "Query Optimization Using Fuzzy Set Theory for Multidatabase Systems," *Proceedings of the 1993 CAS Conference, Volume II*, Centre for Advanced Studies, Toronto Laboratory, IBM Canada, Ltd., 844 Don Mills Road, North York, Ontario, Canada M3C 1V7 (October 1993), pp. 848–859.
29. Q. Zhu and P.-Å. Larson, "Establishing a Fuzzy Cost Model for Query Optimization in a Multidatabase System," *Proceedings of the 27th Hawaii International Conference on System Sciences*, Maui, HI (January 1994), pp. 263–272.
30. D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth, "On Serializability of Multidatabase Transactions Through Forced Local Conflicts," *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan (April 1991), pp. 314–323.
31. H. Garcia-Molina, Y. Breitbart, and A. Silberschatz, "Overview of Multidatabase Transaction Management," *VLDB Journal* **1**, No. 2, 181–239 (1992).
32. J. Veijalainen and A. Wolski, *The 2PC Agent Method for Transaction Management in Heterogeneous Multidatabases, and Its Correctness*, Technical Report J-10, Laboratory for Information Processing, Technical Research Centre of Finland (VTT), Helsinki, Finland (June 1992).
33. A. Wolski and J. Veijalainen, "2PC Agent Method: Achieving Serializability in Presence of Failures in a Heterogeneous Multidatabase," *Proceedings of the IEEE Parbase-90 Conference* (March 1990), pp. 321–330.

34. J. N. Gray and A. Reuters, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann Publishers, San Mateo, CA (1993).
35. Y. Breitbart, A. Silberschatz, and G. R. Thompson, "Reliable Transaction Management in a Multidatabase System," H. Garcia-Molina and H. V. Jagadish, Editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data,* Atlantic City, NJ (May 1990).
36. C. Pu, "Superdatabases for Composition of Heterogeneous Databases," *Proceedings of the Fourth International Conference on Data Engineering* (May 1988), pp. 548–555.
37. J. Slonim, G. K. Attaluri, and P.-Å. Larson, "Advanced Transaction Systems in the CORDS Multidatabase System Environment," *Proceedings of the Workshop on Next Generation Information Technologies and Systems*, Haifa, Israel (June 1993), pp. 130–146.
38. J. N. Gray, R. A. Lorie, G. F. Putzolu, and I. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," G. N. Nijssen, Editor, *Modeling in Data Base Management Systems*, North-Holland, Amsterdam (1976), pp. 365–394.
39. A. M. Joshi, "Adaptive Locking Strategies in a Multi-Node Data Sharing Environment," *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, Catalonia, Spain (1991), pp. 181–191.
40. D. Lomet, *Private Lock Management*, Technical Report CRL 92/9, Digital Equipment Corporation, Cambridge Research Lab, Cambridge, MA (November 1992).
41. S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz, "Ensuring Transaction Atomicity in Multidatabase Systems," *Proceedings of the 1992 Conference on Principles of Database Systems* (1992), pp. 164–175.
42. D. P. Bradshaw, "Open Nested Serializability in Multidatabase Systems," M. Bauer, J. Botsford, P.-Å. Larson, and J. Slonim, Editors, *Proceedings of the 1992 CAS Conference, Volume II*, Centre for Advanced Studies, Toronto Laboratory, IBM Canada, Ltd., 844 Don Mills Road, North York, Ontario, M3C 1V7, Canada (November 1992), pp. 93–109.
43. J. Elliot and B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press Series in Information Systems, The MIT Press, Cambridge, MA (1985).
44. D. Reed, *Naming and Synchronization in a Decentralized Computer System*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA (1978). Also available as Technical Report MIT-LCS-205, Massachusetts Institute of Technology, Cambridge, MA (1978).
45. D. P. Bradshaw, P.-Å. Larson, and J. Slonim, *Transaction Scheduling in Dynamic Composite Multidatabase Systems*, Technical Report CS-94-11, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada (March 1994).
46. D. P. Bradshaw, "Composite Multidatabase System Concurrency Control and Recovery," *Proceedings of the 1993 CAS Conference, Volume II: Distributed Computing*, Centre for Advanced Studies, Toronto Laboratory, IBM Canada, Ltd., 844 Don Mills Road, North York, Toronto, Ontario, Canada M3C 1V7 (October 1993), pp. 895–909.
47. Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz, "On Rigorous Transaction Scheduling," *IEEE Transactions on Software Engineering* **17,** No. 4, 954–960 (September 1991).
48. Y. Raz, *The Principle of Commit Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers*, Technical Report, Digital Equipment Corporation, Maynard, MA (1991).
49. K. Eswaran, J. Gray, R. Lorie, and I. Traiger, "The Notion of Consistency and Predicate Locks in a Database System," *Communications of the ACM* **19,** No. 11, 624–633 (November 1976).
50. H. Kung and J. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems* **6,** No. 2, 213–226 (June 1981).
51. Y. Breitbart and A. Silberschatz, *Complexity of Global Transaction Management in Multidatabase Systems*, Technical Report 198-91, University of Kentucky, Lexington, KY (November 1991).
52. J. Veijalainen and A. Wolski, "Prepare and Commit Certification for Decentralized Transaction Management in Rigorous Heterogeneous Multidatabases," *Proceedings of the Eighth International Conference on Data Engineering*, Tempe, AZ (February 1992).
53. G. K. Attaluri, "Logical Concurrency Control for Large Objects in a Multidatabase System," *Proceedings of the 1993 CAS Conference, Volume II*, Centre for Advanced Studies, Toronto Laboratory, IBM Canada, Ltd., 844 Don Mills Road, North York, Ontario, Canada M3C 1V7 (October 1993), pp. 860–872.
54. *Reference Manual: ObjectStore Release 3.0 for OS/2 and AIX/xlC 300-320-002 3C*, Object Design, Inc., Burlington, MA (January 1994).
55. G. K. Attaluri, "An Efficient Algorithm for Dynamic Indexing of Spatial Objects," to be published in *Proceedings of CASCON '94*, IBM Canada, Ltd., Toronto.
56. G. Attaluri and D. P. Bradshaw, "Architecture for Transaction Management in the CORDS Multidatabase Service," *Proceedings of the 1993 CAS Conference*, Centre for Advanced Studies, Toronto Laboratory, IBM Canada, Ltd., 844 Don Mills Road, North York, Ontario, Canada M3C 1V7 (October 1993), pp. 873–887.

Accepted for publication September 12, 1994.

Gopi K. Attaluri *Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1 (electronic mail: gkattalu@neumann.uwaterloo.ca).* Mr. Attaluri obtained his master's degree in computer science from the Indian Institute of Technology, Kanpur, India, in 1989. He worked in the library automation project at Kanpur during 1989–1990. He has been a Ph.D. student at the University of Waterloo since 1990. He received the ITRC/ICR fellowship at the University of Waterloo. He is currently supported by the Ontario Graduate Scholarship from the province of Ontario, and a student Fellowship from IBM Canada, Ltd. Mr. Attaluri's interests include transaction management, multidatabase systems, and object-oriented database systems.

Dexter P. Bradshaw *Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1 (electronic mail: dpbradsh@bluebox.uwaterloo.ca).* Mr. Bradshaw is a Ph.D. candidate in the Department of Computer Science, University of Waterloo. He holds a B.Sc. degree in

mathematics and computer science from the University of the West Indies, and an M.Math. degree in computer science from the University of Waterloo. He is currently involved in the multidatabase research project at the University of Waterloo, and his current research interests include distributed transaction management, distributed systems, distributed object management systems, and multidatabase systems.

Neil Coburn *Antares Alliance Group Canada, Ltd., Software Development Centre, 2000 Argentia Road, Plaza 2, Suite 2000, Mississauga, Ontario, Canada L5N 1V8 (electronic mail: nzca0@amdahlcsdc.com).* Dr. Coburn completed his Ph.D. in computer science at the University of Waterloo in 1988 and subsequently held the position of research assistant professor in the Department of Computer Science. During that time he worked on several projects sponsored by ITRC (Information Technology Research Centre), NSERC (Natural Sciences and Engineering Research Council of Canada), and CAS (Centre for Advanced Studies) at the IBM Toronto Laboratory. Since 1993 he has worked as a staff systems analyst for the Antares Alliance Group Canada, Ltd. His interests include multidatabases, parallel databases, constraints on complex objects, formalizations of rule-based query optimization, architectures for distributed environments, and the development and maintenance of large software systems.

Per-Åke Larson *Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1 (electronic mail: palarson@bluebox.uwaterloo.ca).* Dr. (Paul) Larson received his Ph.D. from Åbo Swedish University, Finland, in 1976. He joined the Department of Computer Science at the University of Waterloo in 1981 and was promoted to full professor in 1987. He served as chairman of the department during 1989–1992. He was a principal investigator in the CORDS project. His research interests include multidatabase systems, query optimization and processing, and parallel databases.

Patrick Martin *Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada K7L 3N6 (electronic mail: martin@qucis,queensu.ca).* Dr. Martin is an associate professor at Queen's University at Kingston and is a principal investigator in the CORDS project.

Abraham Silberschatz *Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712 (electronic mail: avi@cs.utexas.edu).* Dr. Silberschatz is an endowed professor in the Department of Computer Sciences of the University of Texas at Austin, specializing in the area of concurrent processing. His research interests include operating systems, database systems, and distributed systems. He received the Ph.D. degree in computer science from the State University of New York, Stony Brook, in 1976. He joined the University of Texas at Austin faculty as an associate professor in 1980, earned the rank of full professor in 1984, and received the endowed professorship in 1988. He has been on leave at AT&T Bell Labs since January 1993, where he is currently serving as a senior research scientist. In addition to his academic position, he served as an advisor for the National Science Foundation and as a consultant for several private industry companies. He is active in the field of databases and has served as program committee and general conference chair for a number of symposia in the area. Dr. Silberschatz is a recognized educator, author, and researcher. He is a recipient of the IEEE Computer Society Outstanding Paper Award for the article "Capability Manager" which appeared in *IEEE Transactions on Software Engineering*. He has had articles appear in numerous ACM and IEEE publications, and is coauthor of two well-known textbooks: *Operating System Concepts* and *Database System Concepts*.

Jacob Slonim *IBM Software Solutions Division, Centre for Advanced Studies, Toronto Laboratory, IBM Canada, Ltd., 844 Don Mills Road, North York, Ontario, Canada M3C 1V7 (electronic mail: jslonim@vnet.ibm.com).* Dr. Slonim is head of research at the Centre for Advanced Studies of the IBM Toronto Software Solutions Laboratory. He received his Ph.D. from Kansas State University in 1979. He is an adjunct professor at the University of Waterloo. He served on several international standards committees and has helped to organize numerous international conferences. In 1993, he became a member of the IBM Academy of Technology. Dr. Slonim was project leader for the CORDS project. His research interests include databases, distributed file systems, and software engineering.

Qiang Zhu *Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1 (electronic mail: qzhu@bluebox.uwaterloo.ca).* Mr. Zhu is a Ph.D. candidate in the Department of Computer Science at the University of Waterloo. He holds an M.Sc. in applied mathematics from McMaster University in Canada, and an M.Eng. in computer science and a B.Sc. in mathematics both from the Southeast University in China. He was a principal developer of a relational database management system. He is currently involved in the MDBS research project within the CORDS project. His current research interests include distributed database systems, query optimization, and query processing.

Reprint Order No. G321-5557.