Catalogue management for a multidatabase system using an X.500 directory system

Patrick Martin† and Wendy Powley‡

Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada K7L 3N6

Received 14 June 1996

Abstract. Multidatabase systems (MDBSs) provide applications with integrated access to a collection of databases. The component database systems are typically heterogeneous, distributed, autonomous and pre-existing. MDBSs, like conventional database management systems (DBMSs), require catalogue information to provide their services. The catalogue must be stored in a separate system database, or repository, since component databases are assumed to be independent of the MDBS. We examine the approach taken to catalogue management in the CORDS MDBS which uses an X.500 directory service to store the MDBS catalogue and discuss the advantages and disadvantages of this approach. Storing the MDBS catalogue in a directory service is an appealing approach but is an application for which the directory service was never intended. There are, therefore, practical problems which must be addressed including the suitability of the information model and the performance of the system for typical repository requests. We discuss the design of the CORDS MDBS catalogue and present a set of experiments we conducted with it to examine the performance and scalability of the approach.

1. Introduction

The recent progress in communication and database technologies has fostered both the need and the desire within organizations to access data from a variety of databases and to combine and use the data in new applications that were not anticipated by the original developers of the databases. Thus organizations are now faced with the problems of data connectivity and data integration.

The data connectivity problem is more or less solved. Most modern database systems provide support for remote clients so that an application running on a separate machine can transparently access the database systems. However, the integration problem remains since the application must still deal with different interfaces and must still perform the processing needed to combine the data from the various component database systems (CDBSs).

Integration with respect to database-system interfaces can be achieved where CDBSs support a common interface such as the Microsoft Open Database Connectivity (ODBC) function library [10], the X/Open SQL Call Level Interface [14] or the IBM Distributed Relational Database Architecture (DRDA) [6]. The application still recognizes that it is dealing with multiple databases and is still responsible for the integration processing.

Integration with respect to both interfaces and data processing can be achieved if all the details of how to access the CDBSs are delegated to a separate system which we call a multidatabase system (MDBS). The objective of a MDBS is to provide the application with the view that it is dealing with a single-database system. If a request requires data from multiple CDBSs, the MDBS will determine what data are required from each CDBS, retrieve the data, and perform any integration processing required.

CORDS (COnsortium for Research on Distributed Systems) was a research project that focused on an environment for distributed applications [3]. As part of this project, we participated in the development of a prototype MDBS (called the CORDS MDBS) [1] that provided a single integrated relational interface to a number of CDBSs including relational, hierarchical and network systems.

A MDBS, like any conventional DBMS, must maintain a system catalogue of all the information necessary to operate the MDBS. We adopted a unique approach to catalogue management for the CORDS MDBS by using an X.500 directory service to store the catalogue information. While a number of properties of X.500 favoured its use for this task, questions related to its performance and scalability had to be answered. We discuss the reasons for our approach to catalogue management, comment on...
our experience with the CORDS MDBS and examine the performance and scalability questions by presenting a set of experiments we conducted with the catalogue. The evaluation continues the work initially reported in an earlier paper [8].

The remainder of the paper is organized as follows. Section 2 describes the CORDS MDBS, in general, and, in particular discusses the issue of catalogue management. Section 3 describes the X.500 directory service and presents the design of the catalogue. Section 4 evaluates the potential performance and scalability of our approach. Section 5 summarizes the paper and presents our conclusions.

2. CORDS MDBS

The CORDS MDBS [1], as indicated by the system architecture diagram in figure 1, is a full-function DBMS. The common data model used in the CORDS MDBS is the relational model, so schemas define a collection of data in terms of relational tables and their columns and any applicable constraints. Applications interact with a MDBS server via a library of interface functions called the MDBS client library. A MDBS server performs DBMS functions, such as query processing and optimization, transaction management, and security, at the global level. A MDBS server connects to a CDBS through a server library which accepts SQL requests from the MDBS, interacts with the CDBS through its normal application program interface, and then translates the response into the form expected by the MDBS. The server library, therefore, hides the local data model and interface and presents a relational view to the MDBS server.

A key difference between the CORDS MDBS and a commercial DBMS is its reliance on services provided by the surrounding environment, in particular security services, transaction management services, and a directory service to maintain the MDBS catalogue. We expect that, in a large distributed system, a MDBS would be made up of a number of widely distributed MDBS servers. Each of these servers would manage a set of CDBSs and interoperate with other MDBS servers. Each server could potentially require access to all the catalogue information. The MDBS catalogue, therefore, will have to be logically centralized but physically distributed since a physically centralized catalogue will not provide efficient access to the widely distributed MDBS servers.

The MDBS catalogue stores the data needed by the system during its operation. These data are more complex and extensive than in traditional DBMSs because they must contain information about the schema objects at the MDBS level, schema objects exported by the CDBSs, and the mappings between the two types of schema objects. Figure 2 portrays some of the major entities and relationships in the catalogue data used by the CORDS MDBS.

There may be a number of MDBS servers in a
distributed system. Each server provides service to applications and accesses a number of sites. Each site holds one or more CDBSs. Access to the data of an application is through a MDBS schema which specifies table, column, and constraint definitions. Requests against a MDBS schema may be compiled and stored in the form of access plans for subsequent execution. Any errors detected by a CDBS during the processing of a request must be reported in terms of the set of the MDBS errors. The data available from a CDBS are defined in terms of one or more export schemas. Each export schema is a relational representation of the data available from a CDBS and contains definitions of the available tables, columns, constraints, and indexes.

The catalogue data are accessed in a number of ways depending upon the system component performing the access. For example, in the CORDS MDBS, system run-time components, such as the parser, optimizer and view integrator, issue queries for specific objects, typically using a system identifier, and perform updates on run-time information. Short response times for accesses by the run-time components are important for acceptable system performance. Users also need the capability to explore the data by querying and by browsing, and to issue updates to the information.

The mappings between corresponding MDBS schema objects and export schema objects are provided during the schema integration process. We do not define a single all-encompassing global schema but instead define different MDBS schemas to provide the data for individual applications, or groups of applications. MDBS schemas are equivalent to the federated schemas defined by Sheth and Larsen [13]. MDBS schemas are made up of virtual relations called MDBS views that may span multiple heterogeneous CDBSs. MDBS views are similar to relational views in that they are not materialized but rather are stored as mappings which are invoked whenever a MDBS view is accessed.

Our current syntax for MDBS views extends the standard SQL view definition facility with support for attribute contexts and transformation functions [9]. Attribute contexts are used to describe the semantics of attributes used and transformation functions are used to resolve conflicts among the set of export attributes contributing to a MDBS-level attribute.

2.1. Example MDBS schema

To illustrate MDBS views, consider an example application which accesses data about publications from multiple sources. One data source, described by the export schema UWBooks, represents books at the University of Waterloo and is shown in figure 3. A second data source, shown in figure 4, contains data about publications at Queen’s University and is described by the export schema QueensPubs. The attributes which form the primary key of each relation are specified as not null. To simplify the example, we assume that the values in corresponding columns in the export tables are drawn from the same domain so conflict resolution is not required.

An application at the University of Western Ontario accesses the two data sources through the MDBS schema.

```
CREATE MDBSView
(
    Export DATABASE UWBooks (EXPORT TABLE Publisher (PubNo integer not null, Name char(40), Address char(40)) FROM uwaterloo.CDBS1.Publisher
    EXPORT TABLE Book (ISBN integer not null, Title char (40), PubNo integer) FROM uwaterloo.CDBS1.Publisher
    EXPORT TABLE Author (ISBN integer not null, AuthorName char(40) not null, Affiliation char (20)) FROM uwaterloo.CDBS1.Publisher
    )
)
```

**Figure 3.** Export tables for UWBooks.

```
CREATE MDBSView
(
    Export DATABASE QueensPubs (EXPORT TABLE Items (ISBN integer not null, Title char(40), Publisher char(40), PubAddress char(40), Type char(10)) FROM queensu.CDBS2.Items
    EXPORT TABLE Authors (ISBN integer not null, Surname char (20) not null, FirstName char (20), Affiliation char(20)) FROM queensu.CDBS2.Authors
    )
)
```

**Figure 4.** Export tables for QueensPubs.

*WesternOntarioBooks* which contains one MDBS view *Books* shown in figure 5. The view combines information on books from both the Queen’s University and University of Waterloo databases.

Processing the *Create MDBSView* statement results in information about the MDBS view, and the mappings between *Books* and the exported tables being stored in the MDBS catalogue. The MDBS uses this information to translate and decompose the application’s SQL requests, which are posed against *Books*, into appropriate queries on the CDBSs.

3. Catalogue management in the CORDS MDBS

The approach used in the CORDS MDBS, as stated earlier, is to store the catalogue data in an X.500 directory service. We chose this unique approach to catalogue storage and management for the following reasons.

1. Discussions with other research groups in CORDS, in particular the group working in the area of systems
management, led to the conclusion that there was a need to store and manage common information and that this sharing of information could be exploited by both groups. The use of the directory service was already being explored by the systems-management group, thus X.500 was already an integral part of the overall CORDS architecture. There exist many other similar services that could have been utilized to implement the MDBS catalogue such as a replicated relational database system or another protocol. However, the systems-management group, thus X.500 was already an integral part of the overall CORDS architecture. There exist many other similar services that could have been utilized to implement the MDBS catalogue such as a replicated relational database system or another protocol such as WHOIS++ [4], but because X.500 was part of the CORDS environment and it appeared to provide the services required by the MDBS catalogue, it seemed natural to explore the use of X.500 for this purpose.

(2) While the most common use for a directory service is for locating objects and services by name, it is effectively a distributed database, which can potentially be used to store any type of information. Thus the directory service may be used to implement a catalogue that is logically centralized, physically distributed and at least partially replicated.

(3) The X.500 service supports an object-oriented model with inheritance and aggregation and provides the capability to define new attribute types when required. Thus the data model is rich enough to represent the entities and relationships that make up the catalogue data.

(4) X.500 is an internationally accepted standard with a large number of systems already in use which would simplify the task of moving the MDBS into an open, widely distributed environment.

The two main tasks in developing the catalogue were to define the X.500 schema to hold the catalogue data and to develop an API for the MDBS components to use in accessing the catalogue. We first present an overview of the X.500 standard and then discuss the design of the X.500 schema and the catalogue API in the remainder of the section.

3.1. X.500 directory service

The X.500 directory standard [5] specifies a directory service that provides and manages information about communication entities. These entities are represented by entries in a hierarchical name space, the X.500 directory information tree (DIT). Entries are usually placed in the DIT according to the organizational relationships between the real-world entities that they represent.

A DIT entry contains information about the entity that it represents. This entity information is in the form of attributes whose types and structures are governed by a set of rules, or integrity constraints, called schemas. Each entry in the DIT is labelled by a relative distinguished name (RDN) composed from an attribute or a set of attributes of that entity and which is unique among the other entries that are children of the same parent entry. In this way, each entry has a globally unique name that is composed of the concatenated sequence of RDNs in the path from the root of the DIT to the entry. This globally unique name is the entry’s distinguished name (DN).

The X.500 abstract service definition defines abstract ports and operations that provide the user functionality for retrieving, searching and modifying directory information. These operations form the directory access protocol (DAP). The current X.500 standard supports the read, compare, list, search and abandon interrogation functions and the basic manipulation functions—add, remove, modify entry and modify DN. However, the directory service’s modification functionality is limited in that it does not support the arbitrary creation and deletion of non-leaf entries.

The directory service (including the DIT) is distributed over physically separated entities (such as computer nodes) called directory system agents (DSAs). The distribution is transparent to the user (when chaining rather than referrals is used). The set of entities and information managed by a DSA is called its directory information base (DIB). Each user or user-process is represented by a directory user agent (DUA) that is responsible for querying or interactively interrogating the directory. The DUA acts as a client to the directory.

The DUAs are provided access to the DIT through DSAs; each DUA accesses only a few of the DSAs, typically one. The DSAs may communicate with each other in order to satisfy requests from the DUAs. The DAP defines the requests and responses between a DUA and a DSA, while the directory system protocol (DSP) defines the requests and responses between communicating DSAs.

3.2. Catalogue schema design

The X.500 directory schema defines the set of rules and constraints concerning the structure and relationships between objects within the DIT. The schema includes definitions for object classes, attributes, and attribute syntax. The X.500 directory service is typically used to store information about communication entities, so object classes and attributes associated with these entities are standard but several new object classes and attributes had to
be specified to represent the MDBS metadata. The X.500 DIT structure corresponding to the ER schema in figure 2 is shown in figure 6.

In many cases, the translation from the ER model to the X.500 model is simple. An entity in the ER schema is mapped to an object class in the X.500 schema, and the attributes are modelled by equivalent X.500 attributes. For instance, the site, CDBS and export table entity types in the ER schema are mapped directly to the site, CdbS and ExpTable X.500 object classes, respectively.

Some relationship types in figure 2 are represented by attributes of an object class in the X.500 schema. For instance, the many-to-many relationship MappedTo between MDBS column and export column is represented by an attribute, sourceColumnOid, of class MdbsColumn in the X.500 schema. The sourceColumnOid attribute is a list of export column identifiers which define the MDBS column.

3.2.1. Examples revisited. To illustrate how the definitions are used and how the catalogue data are stored in the directory service, we use the shared library example specified earlier in the paper. A DIT for this example might be structured as in figure 7. Here, we let mA denote an MDBS application (MdbsApp), cD denote component database (Cdbs), mS denote an MDBS schema (MdbsSchema), eS denote an export schema (ExpSchema), mT denote an MDBS table (MdbsTable), mC denote an MDBS column (MdbsColumn), eT denote an export table (ExpTable), and eC denote an export column (ExpColumn). The two upper layers of the DIT follow a standard pattern in X.500 which groups organizations (‘O = UWO’, ‘O = UW’ and ‘O = Queens’) under their host country (‘C = Canada’). Note that figure 7 depicts only a small portion of the entries stored for the library example.

The entries defining the University of Waterloo CDBS, the export schema, the table Book and the column ISBN might be as shown in figure 8.

The entry definition of the component database, CDBS1, specifies the host on which it resides, the export schemas from that source and other optional information, such as specific product databases. In this example, this information is empty. In the actual directory information base these attributes would be absent altogether; the directory schema definition specifies all possible attributes and the specific entries may or may not have values for the optional attributes. Mandatory attributes must be present at the time an entry is created.

The definition of the export schema UWBooks specifies its export tables. Although these are presented as elements of a set, each is defined as a separate attribute/value pair within the directory information base. Thus a request to the directory for the tables associated with UWBooks would result in all export table attribute/value pairs being retrieved. The export columns defined within for the entry book would behave similarly.

The definition of the entry representing the multi-
Entry CDBS1 {
/* Mandatory attributes */
cdbsName="CDBS1",
objectClass=Cdbs,
/* Optional attributes */
siteName="@C=Canada@O=UW@site=bluebox",
exportSchemaName="@C=Canada@O=UW@site=bluebox@cd=CDBS1@es=UWBooks",
cdbsType="",
cdbsProduct="",
remoteJoinFlag="n",
semiJoinFlag="y",
description="University of Waterloo library database.")
}

Entry UWBooks {
/* Mandatory attributes */
exportSchemaName="UWBooks",
objectClass=ExpSchema,
/* Optional attributes */
timestamp="",
description="Library books export schema"
}

Entry Book {
/* Mandatory attributes */
tableName="Book"
objectClass = ExpTable,
/* Optional attributes */
noOfColumns=3,
noOfIndexes=2,
description="Author table in UWBooks"
}

Entry ISBN{
/* Mandatory attributes */
columnName="ISBN",
objectClass = ExpColumn,
/* Optional attributes */
domain="isbn-syntax-string",
nullValuesFlag="n",
primaryKeyFlag="y",
foreignKeyFlag="y"
}

database application and its schema might be defined as in figure 9. When the SharedLibrary application is executed, the MDBS client for the application can retrieve information about the application’s schema, namely WesternOntarioBooks, the associated table and column definitions for the schema, and the definitions of the contributing CDBSs and export schemas.

Figure 8. X.500 export schema objects.

Entry SharedLibrary{
/* Mandatory attributes */
mdbsApplicationName="SharedLibrary",
objectClass=MdbsAppl,
/* Optional attributes */
hostSite="@C=Canada@O=UWO",
description="External selection of books"
}

Entry WesternOntarioBooks{
/* Mandatory attributes */
MdbsSchemaName="WesternOntarioBooks",
/* Optional attributes */
componentDataSource="@C=Canada@O=UW@site=bluebox@cd=CDBS1",
componentDataSource = "@C=Canada@O=Queens@site=abbott@cd=CDBS2",
timestamp="",
description="Schema defining external library books"
}

Figure 9. X.500 MDBS schema objects.

3.3. Catalogue API
The MDBS catalogue is accessed by many components of the CORDS MDBS, including the parser, the optimizer, the plan manager, the view integration system and the catalogue browser. A catalogue access involves either an update (adding information to the catalogue) or a retrieval (obtaining information from the catalogue).
Each catalogue access must be preceded by a bind operation to connect to the DSA. Because the bind operation can take several seconds to perform (at least in the version of X.500 used for our testing), a single bind operation is performed when a MDBS server is started. When the server is shut down, the connection is terminated.

In the current prototype, two modules provide access to the catalogue. One module, the DUA, includes functions which provide direct access to the X.500, including bind, unbind, read, search, add and modify. On top of this module, we have implemented the catalogue API. This module provides specific access routines needed thus far by the various parts of the MDBS. These are divided into two categories: update functions and retrieval functions.

**Update functions.** Catalogue updates involve either adding a new entity to the catalogue or modifying an existing entry. Currently, updates only occur when an export table or an MDBS table is defined. The MDBS parser processes the request and then calls the appropriate update functions from the catalogue API to add the new entries to the catalogue. Several new catalogue entries may result from a single query. For example, an export table definition may require entries for the site, the component database, and the export schema as well as for the export table and each of its attributes.

The update functions are responsible for compiling the information about a new object and calling the DUA Add function to add the entry to the DIT. Each entry is added to the catalogue separately and sequentially with superior entries being added prior to their children. For example, the export table is added to the catalogue before the export table columns, as the table entry is the superior entry for the column.

Catalog updates will typically involve modifying several directory entities within the one logical update, or may even involve updating objects on multiple DSAs. A transaction mechanism is therefore required to ensure the atomicity of the set of X.500 updates but transactions are not part of the X.500 standard. However, research efforts, such as Neufeld [11], have incorporated transactions into X.500 implementations so we assume that transactions can be made available.

**Retrieval functions.** The retrieval functions allow components to access the various pieces of information stored in the MDBS catalogue. The X.500 directory service provides functions for basic types of queries, however the MDBS requires more complex access functions such as a series of searches and reads. These have been implemented as a set of library functions that can be used by the various MDBS components.

X.500 is designed for efficient access to an entity based on its DN. Several components of the MDBS, however, rely on a unique object identifier assigned by the MDBS parser and generate frequent requests for information about an object based on the object identifier rather than its DN. The object identifiers are stored as part of the catalogue information, so it is possible to locate an object by using an appropriate search filter but such a search is inefficient since it is not based on DN and may involve searching the entire DIT.

To improve the efficiency of retrievals based on object identifiers, we use a translation table to store mappings between DNs and object identifiers. The translation table is stored as an entity in the X.500 directory and is read into main memory as part of the initialization process of a MDBS server. To obtain information about an entity using an object identifier, the translation table is first consulted to obtain the DN of the entity, which is then used to retrieve the information with a single X.500 catalogue read operation.

Another possible approach to retrieval based on object identifier, which is totally within X.500, is to create a special subtree of aliases based on the object identifier. However, experiments with this approach in another context (see Barrowman and Martin [2]) indicated that retrieval time using aliases could be significantly longer than direct reads using DN. Since there is a high frequency of retrievals based on object identifiers, the translation table is a better approach.

The retrieval functions are typically of the form ‘given X, return Y’, where X and Y are generally names (but not usually DNs) of objects and/or object identifiers. The translation table is used whenever possible to facilitate information retrieval and to reduce the number of catalogue accesses. The DN of an object, for instance, can be used to provide several pieces of information about the object without a single catalogue access. For instance, the DN of an export schema contains the name of its host site and its CDBS. By parsing the DN, we can determine this information without having to access the catalogue.

The retrieval functions have been implemented on an ‘as-needed’ basis. As the MDBS system expands, additional access functions will be required. An alternative to developing individual functions to retrieve specific pieces of information is to provide a general query interface to the catalogue which will provide support for more complex queries as well as for catalogue browsing (for example see [2]).

### 4. Evaluation of performance and scalability

An approach based on an X.500 directory service has a number of advantages—the most significant of which is its distributed database capabilities. We are, however, using the directory service for an application for which it was not originally intended, and there are concerns regarding the performance and scalability of the approach. To provide some insight into these issues, we present three sets of experiments which reveal the performance of the catalogue for its three main types of accesses: updates (a new catalogue entry is added), retrieval by DN and retrieval based on other attribute values.

All tests were performed on a suite of IBM RS/6000 Model 220 computers each with 32 MB of RAM. The catalogue was based on the ISO/IEC Q10P (version 8.0) [12] implementation of X.500. Each test was run a total of 10 times and the average time calculated. The reported times include only the actual time spent accessing the DSA.
and do not include the DUA time or time required for the catalogue API functions. The catalogue was stored in an application specific, private, non-indexed DIT and X.500 caching was disabled. Thus we should note that the configuration used was not set to optimize retrieval performance.

To get an idea of how well these operations will scale up as the size of the MDBS increases, we ran all tests on multiple sample sizes. In addition, we wanted to determine the effects of distributing the MDBS catalogue information among multiple DSAs. Therefore, we ran the tests using a single DSA, two DSAs, three DSAs and four DSAs. DSAs were physically separated (that is, each on a different machine), but were within the same local network.

It is still unclear exactly how the MDBS catalogue information will be split among multiple sites, so for testing purposes we assumed the worst case, that is all catalogue accesses were to the most remote DSA. Multiple DSAs were created: the local, or top-level DSA and one, two or three subordinate DSAs. The local DSA contained no MDBS catalogue information, only a reference to a subordinate DSA. Likewise, in the case of three and four DSAs, the intermediary DSAs contained no MDBS information, only a reference to another DSA. In each case, all participating DSAs were contacted in order to access the DSA containing the MDBS information. All remote bindings were established prior to measurement. We note that the retrieval times are pessimistic since we had to disable caching within the directory system to ensure consistent results.

To provide a reference point for the evaluation of our approach, we also implemented the catalogue using the IBM DB2/6000 relational database management system [7]. The relational schema shown in the appendix was defined based on the entities and relationships shown in figure 2. The schema corresponds closely to the X.500 schema. In most cases, X.500 classes became tables and the attributes were represented as columns. The catalogue API was implemented using the DB2 call level interface (CLI). A corresponding set of tests were run on the relational version of the catalogue on the same hardware for comparison with the X.500 catalogue. In this case, since our version of DB2 did not support distributed databases, the catalogue was not distributed.

4.0.1. Updates. To estimate the time required to add new entries to the catalogue, we measure the average time to add a ‘typical’ export-table definition. An export-table definition is defined (for experimental purposes) to be an export table name and five attributes which results in a total of six catalogue entries. These are performed separately and sequentially. We ensure that the site, CDBS and export schema (which must be added before an export table) are present in the catalogue before the export table definition is added. The total time required to add the six entries making up the export table definition is reported in table 1. Two sample sizes are used: 100 and 400 export-table definitions. The version of X.500 used for the catalogue did not permit chained updates, that is updates to a remote DSA, due to potential security risks so the update tests were run on a single DSA only.

Table 1 shows that updates to the X.500 catalogue are slow relative to updates to the DB2 catalogue. Although the time to add entries increases as the catalogue size increases for the X.500 implementation, this increase is less than linear. It is expected that updates to the MDBS catalogue will be relatively infrequent and will not be time-critical.

4.0.2. Retrieval based on distinguished name. X.500 is designed for efficient retrieval of information using an object’s DN. An object’s DN is similar to the primary key of a table in that it uniquely identifies the object within its domain (that is, the DIT for X.500; a relation for DB2). This test compares the time required to retrieve a piece of information from the X.500 catalogue using the DN to the time required to retrieve information from a single table in the relational catalogue using the primary key. In both cases, we expect this to be the most efficient access method.

In the X.500 catalogue we used the X.500 READ command to retrieve each object’s object identifier (OID). This command locates the object using its DN, then returns the desired attribute value (in this case, the OID). A similar test was run using the DB2 catalogue. OID is the primary key of each table in the DB2 catalogue. For each catalogue object, we retrieve the object’s name from the appropriate table using its OID. The SQL query is of the form

\[
\text{select Name from TableName where OID = XXX}
\]

where ‘TableName’ is the table where the object resides and ‘XXX’ takes on the value of each object identifier in the catalogue. Three catalogue sizes were used—approximately 100, 400 and 2400 entries†.

Basic retrievals using the DN of the object are expected to be the most-frequent operation performed by the internal modules of the MDBS. Due to the relatively flat nature of the MDBS schema hierarchy, we expect that the time to access a single object will be short and should remain constant as the catalogue size increases. The results in table 2 show that for a single DSA, these assumptions are correct. The X.500 catalogue performs nearly as well as the DB2 catalogue. When the information is distributed among multiple DSAs, the performance of the X.500 catalogue declines both as the number of DSAs increases and as the catalogue size increases. The size of the decrease caused by distribution, however, is not that great considering that we are looking at a worst-case performance. The results suggest that retrieval based on DN is currently efficient and will scale-up satisfactorily.

† These numbers are given as approximate since the number of entries varied slightly between the X.500 and relational DBMS catalogue implementations.
4.0.3. Retrieval based on attribute value. This test measures the length of time required to search and retrieve information based on a particular attribute value. In the X.500 catalogue, the X.500 SEARCH command is used which takes as parameters the base object (which specifies where in the DIT to start the search), a filter, and the name(s) of the attribute(s) to be returned. The subtree below the base object is searched for objects that satisfy the filter. We use the root of the DIT as the base object so the entire tree is searched with each query. The filter used is ‘OID = XXX’ where ‘XXX’ takes on the value of each object identifier in the catalogue. For simplicity, we retrieve the object identifier (since this is the only attribute common to all objects).

A similar test is performed using the relational catalogue implementation. For each object in the catalogue, we retrieve the object identifier given the object’s name. The SQL query is of the form

select OID from Table where Name = XXX

where ‘XXX’ takes on the name of each object (names are unique within tables) and ‘TableName’ indicates the table in which the object is located. To satisfy such a query, an entire table must be examined. As in the previous experiments three catalogue sizes were used—approximately 90, 400 and 2400.

Table 3 shows that searches based on attribute value are less efficient than searches based on DN (or primary key in the case of the relational catalogue). Comparing a single DSA and DB2, we see that X.500 actually performs better in this case. This is surprising since a search in X.500 requires the entire DIT to be examined whereas in DB2 only a single table is searched. As was expected, search time increases quite dramatically as the catalogue size increases. Also, distribution causes a decline in overall performance.

Searches based on attribute value might be used during the schema integration stage for such tasks as finding similar items within the catalogue. As the integration is not time critical, the fact that these searches require more time to complete is not as crucial.

5. Conclusions

Multidatabase systems are an important new type of database system which provides integration of, and interoperability among, a collection of legacy database systems. The CORDS MDBS is a prototype that provides a single integrated relational interface to legacy systems including relational, hierarchical and network DBMSs. A MDBS, like any conventional DBMS, must maintain a catalogue of all the information necessary to operate the MDBS. In the CORDS architecture, the MDBS is made up of a number of cooperating MDBS servers. The MDBS catalogue, therefore, must be available to a set of widely distributed MDBS servers. The MDBS catalogue must also contain a wider variety of data than conventional DBMS catalogues. In particular, it must contain descriptions of data objects at both the CDBS and MDBS levels, and it must contain the mappings between the levels to provide schema integration.

In this paper, we discuss our experiences using an X.500 directory service to provide catalogue management for the CORDS MDBS. We chose to use the directory service for several reasons. First, the directory service provided a way to share data with the subsystems providing systems management in the CORDS environment. Secondly, the catalogue data had to be physically distributed and the directory service is a distributed database. Thirdly, the X.500 service supports an object-oriented model rich enough to represent the entities and relationships that make up the catalogue data. Finally, X.500 is an internationally accepted standard with a large number of systems already in use which would simplify the task of moving the MDBS into an open, widely distributed environment.

In order to use X.500 to implement the MDBS catalogue we had to address two main issues. We had to design a schema for the catalogue data in the X.500 data model, and we had to construct an appropriate API to the MDBS catalogue for the otherMDBS server components. The paper presents a conceptual schema for the catalogue data and then shows how that conceptual schema is mapped to an X.500 class hierarchy. We found that the different notions of hierarchy in the two models complicated the mapping process. In X.500 the hierarchy reflects the naming structure while in the conceptual model the hierarchy is related to generalization.

The paper also describes a catalogue API suitable for the other components of the MDBS server which hides many of the details of the X.500 implementation. The API consists of operations to both retrieve catalogue data and update catalogue data. One problem we found was that while X.500 is designed for efficient access to an entity based on DN, the majority of retrievals by MDBS server components were based on an internal object identifier. We described our solution to this problem which involves maintaining a table within the catalogue interface to translate between object identifiers and DNs so that the most efficient type of accesses to X.500 would be used. The DN itself can also provide information to the MDBS components since it is constructed in a hierarchical manner from the names of containing objects. So some queries
Table 3. Retrieval based on attribute value.

<table>
<thead>
<tr>
<th>Catalogue size (number of entries)</th>
<th>Average time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 DSA</td>
</tr>
<tr>
<td>100</td>
<td>0.11</td>
</tr>
<tr>
<td>431</td>
<td>0.18</td>
</tr>
<tr>
<td>2405</td>
<td>0.64</td>
</tr>
</tbody>
</table>

can be answered from the information in the translation table therefore reducing the number of calls to the directory service.

The working prototype system that was developed by the CORDS MDBS project proved the feasibility of our approach to catalogue management. However, questions concerning the performance and scalability of the MDBS catalogue remained. The paper described a set of experiments which investigated these two issues.

Our experiments showed that updating the catalogue can be a problem. Updating the X.500 catalogue is slow relative to the DB2 catalogue. The time to update the X.500 catalogue also increases as the size of the catalogue increases so the approach may not scale up. The QUIPU directory service has limitations that affect updating, namely that transactions are not supported and chained modifications to other DSAs are not allowed. One saving point is that updates to the catalogue will be much less frequent than retrievals.

We found that the performance of retrievals based on DN for one DSA are comparable to retrievals from DB2 based on the primary key. There was also no significant increase in retrieval time for one DSA as the size of the catalogue increased. Including more DSAs in a retrieval increases the time required by approximately two to three times for each new DSA. We note that, as mentioned earlier, caching was disabled so the retrieval times for multiple DSAs are pessimistic. We conclude that as long as most queries can be answered by one DSA then performance and scalability will be satisfactory which implies the need for careful data placement and the use of replication within the catalogue.

For retrievals based on attribute values, the size of the catalogue has an impact on performance of both the X.500 and the DB2 catalogues. The X.500 catalogue has better performance than the DB2 catalogue for most cases tested except when the catalogue is large and more than one DSA is involved in the query. So while the results for the X.500 catalogue are not unreasonable when compared with the DB2 catalogue, the performance trends indicate that the X.500 will not scale up well for this type of retrieval.

In summary, our experiments indicate that, for the types of accesses most used in the MDBS, the X.500 catalogue is a viable approach with respect to performance and scalability when compared with the DB2 catalogue. Thus, we conclude that the cost of acquiring the openness and distributed database features of X.500 are reasonable and the approach is worth further study.

Acknowledgments

This research is supported by IBM Canada Ltd and the National Science and Engineering Council of Canada.

Appendix: Relational schema for DB2 catalogue experiments

create table AppColumn (  
RDN char(40) NOT NULL,  
OID char(40) NOT NULL,  
appDomainOID char(40),  
nullValuesFlag char(40),  
primaryKeyFlag char(40),  
foreignKeyFlag char(40),  
description char(100),  
AppTableOID char(20),  
umOfExportCols char(20),  
exportColumnIds char(20),  
domain char(40),  
int1 char(10),  
int2 char(10),  
units char(80),  
defaultValue char(20),  
numberInSet char(10),  
DomainSet char(150),  
minValue char(10),  
maxValue char(10),  
primary key (OID))

create table AppTable (  
RDN char(40) NOT NULL,  
OID char(40) NOT NULL,  
schemaOID char(40),  
noOfAttributes char(40),  
parseTree BLOB(200),  
SQLDefinition long varchar,  
description char(100),  
umOfExportTables char(20),  
exportTableIds char(20),  
primary key (OID))

create table CDBS (  
RDN char(40) NOT NULL,  
OID char(40) NOT NULL,  
siteOID char(40),  
description char(100),  
cdbsType char(40),  
cdbsProduct char(40),  
tempStorageFlag char(40),  
buildIndexesFlag char(40),

212
remoteJoinFlag char(40),
semiJoinFlag char(40),
maxFlag char(40),
minFlag char(40),
sumFlag char(40),
avgFlag char(40),
countFlag char(40),
sortFlag char(40),
groupFlag char(40),
noOfUsers char(40),
nOOfUsersTimestamp char(40),
primary key (OID))

create table ExportColumn (  
RDN char(40) NOT NULL,  
OID char(40) NOT NULL,  
exportTableOID char(40),  
exportDomainOID char(40),  
nullValuesFlag char(40),  
minValue char(40),  
maxValue char(40),  
description char(100),  
int1 char(40),  
int2 char(40),  
defaultspec char(40),  
units char(40),  
defaultValue char(40),  
numberInSet char(40),  
domainSet char(40),  
primary key (OID))

create table ExportSchema (  
RDN char(40) NOT NULL,  
OID char(40) NOT NULL,  
cdbsOID char(40),  
description char(100),  
timestamp char(40),  
primary key (OID))

create table ExportTable (  
RDN char(40) NOT NULL,  
OID char(40) NOT NULL,  
schemaOID char(40),  
nOOfAttributes char(40),  
nOOfIndexes char(40),  
nOOfTuples char(40),  
nOOfTupleTimestamp char(40),  
avgTupleSize char(40),  
avgTupSzTimestamp char(40),  
SQLDefinition char(254),  
description char(100),  
primary key (OID))

create table MDBS (  
RDN char(40) NOT NULL,  
OID char(40) NOT NULL,  
MDBAOID char(40),  
description char(100),  
primary key (OID))

create table Site (  
RDN char(40) NOT NULL,  
OID char(40) NOT NULL,  
siteAddress char(40),  
procType char(40),  
procProduct char(40),  
noOfProc char(40),  
procSpeed char(40),  
memSize char(40),  
cacheSize char(40),  
avDevRetRate char(40),  
avDevCap char(40),  
OS char(40),  
optSoftware char(40),  
login char(40),  
avgLoad char(40),  
description char(40),  
primary key (OID))

create table Translation (  
RDN char(40) NOT NULL,  
OID char(40) NOT NULL,  
type char(20),  
primary key (OID))

References

[14] X/Open 1992 Data Management: SQL Call Level Interface (CLI) X/Open Company Limited, California