# The Performance of SQL Queries to an X.500 Directory System*

November 1995
External Technical Report
ISSN 0836-0227-394

David Barrowman and Patrick Martin
Dept. of Computing and Information Science
Queen's University at Kingston
Kingston, Ontario
CANADA K7L 3N6
email: {barrowma, martin}@qucis.queensu.ca

November 30, 1995

**Abstract**

The X.500 standard specifies a distributed directory service designed to store information about people and objects associated with computer networks. Its API is geared toward retrieving information based on this application domain. Recently, a number of projects have used the directory in non-traditional ways. Such applications, however, are constrained by the X.500 information model and the limited functionality of its API. We describe a prototype system that allows users to view the information in a directory as relations, and to query the view using SQL. An analysis of the performance of the system is presented and possible optimizations are discussed.

## 1 Introduction

The X.500 Directory Service was designed to act as a distributed white pages service to store information about telecommunications entities. Increasingly, however, X.500 directories are being used to store more general types of data, for example multidatabase catalog information [2] and systems management information [7]. While applications such as these enjoy the benefits of the

*The authors thank IBM Canada Ltd. and NSERC for their support of the research.

distributed nature of the Directory Service, they are constrained by its information model and API, which were designed with traditional X.500 applications in mind. Particularly limiting, in this respect, are the inability of the X.500 information model to represent the arbitrary relationships that may exist between the objects being modelled, and the consequential inability of the X.500 API to query those relationships.

Relational database management systems are currently the most popular form of general database system. The relational model is capable of representing most types of business data and all relational systems provide applications with the ability to query the database using SQL, the accepted standard database query language. SQL allows ad hoc queries which combine information from multiple tables to be described unambiguously. A relational model interface to X.500 could provide new applications which use the Directory in non-traditional ways with the benefits of both the power and flexibility offered by SQL and the transparent distribution offered by X.500. Furthermore, such an interface would allow traditional X.500 directories to be accessed through a well known database query language.

This paper describes a prototype relational query processor for an X.500 directory system. We analyze the performance of a set of relational queries to the directory and suggest a number of query optimization techniques based on the analysis. The remainder of the paper is organized as follows. Section 2 sets the research into context and provides background information. Section 3 presents the mapping, from X.500 to the relational model, that was chosen as the basis for our SQL interface and briefly describes the query processing algorithm implemented in the prototype. Section 4 presents a performance analysis of our query processor. This includes a description of the benchmark databases and queries, and a discussion of the results obtained. Section 5 discusses some of the techniques which were examined in an effort to optimize the system. Finally, Section 6 presents our conclusions.

## 2 Background

We set the context for the research discussed in the paper by briefly examining three topics. First, we describe the X.500 Directory Service. Second, we discuss two previous attempts at linking X.500 and relational query languages and finally, we present an application of the X.500 Directory Service which motivates the research.

## 2.1 X.500 Standard

X.500 [8] is an International Telecommunications Union (ITU) standard which specifies a Directory Service for managing information related to communication entities. The most noticeable feature of an X.500 Directory is that while its clients are presented with a single, logically centralized view of the information, that information can actually be physically distributed and replicated. Results of this property are improved availability of the Directory Service within a distributed environment, but reduced performance (as compared to that of a single, centralized repository service), due to the required increase in network traffic.

### 2.1.1 The Directory Information Base

The Directory Information Base (DIB) is the set of information stored in the Directory. The DIB contains entries which describe objects in terms of their attributes. (An attribute consists of a type and one or more values.) These objects are organized as nodes in a tree-structured hierarchical name space, termed the Directory Information Tree (DIT). The placement of the objects in the DIT is based on the real-world organizational relationships that exist between the objects they represent. Each object has an attribute, or set of attributes, termed its Relative Distinguished Name (RDN), which differentiates it from all other objects with the same parent object in the DIT. Thus, an object is uniquely identified in the DIB by its Distinguished Name, which is simply the concatenation of all RDNs that are found along the path from the root of the DIT to the particular object's entry. X.500 also allows aliases to be defined, so that a single object may be associated with multiple parent nodes in the DIT.

Each X.500 object has an object class. This is used to identify what type of real-world entity the object represents, and determines which attributes are present in its Directory entry. A number of basic object classes, all of which pertain to communication entities, are defined in the X.500 Standard. However, the Standard also allows for user defined classes, which promotes extensibility, and permits the directory to be used in non-traditional ways.

The information in the Directory is physically distributed across a number of Directory System Agents (DSAs). When a user process wishes to access the Directory, it interacts with a Directory

User Agent (DUA). From the user's point of view, it appears that the DUA simply accesses the information from a centralized directory. However, in order to satisfy the client's request, the DUA interacts with individual DSAs, which, in turn, collaborate to find and return the necessary information. This process allows the benefits of distribution to be realized, while maintaining the transparency of that distribution.

### 2.1.2 Searching The Directory

The Directory Access Protocol (DAP), which is used by DUAs to communicate with the Directory, provides the functionality required to search for, retrieve, and modify the information stored in the Directory. Included in the current DAP specification are the *Read*, *Compare*, *Search*, *List*, and *Abandon* functions used to interrogate the DIB, and the *Add*, *Remove*, *Modify Entry*, and *Modify RDN* functions used to alter the contents of the DIB.

The *Search* function allows a user to query an X.500 database. Among the arguments that it takes are a filter, a base object, and a subset parameter. The filter specifies attribute values that an object must have in order to qualify as a result of the search. A variety of types of attribute value comparisons, such as equality and substring matching, are permitted in the filter. The base object parameter specifies the object at which the search should start, thereby restricting the search to a subtree of the DIT. The search can also be narrowed through the use of the subset parameter, which allows the user to limit the search to the base object, to the immediate children of the base object, or to the entire subtree of the DIT rooted at the base object. The benefit of focussing the search of the DIT by supplying a base object and a subset value is that only relevant portions of the DIT are searched, thereby reducing the number of objects that must be examined to satisfy a query; the drawback is that the user must have a knowledge of the structure and contents of the DIT.

A number of working implementations of the X.500 Directory Service Standard exist; the implementation used in this project is QUIPU 7.0 [11]. QUIPU offers a procedural API and a number of end user interfaces, such as the Directory Shell (DISH).

## 2.2 Previous Work

As opposed to the X.500 DAP and its *Search* function, SQL and its familiar *Select-From-Where* construct is highly flexible. Especially important is its ability to join information from multiple relations in a database, thus promoting querying of the relationships that exist between the objects being modelled. AID [1] and Nomenclator [10] are two previous attempts to integrate X.500 with relational query languages.

AID implements an SQL-like interface to QUIPU. Its queries have the *Select-From-Where* form, but the From clause may only specify a single table. Each node in a DIT is considered to be a table; the tuples belonging to a table are composed of the attribute values of the objects at and below its corresponding node. This introduces the problem of tables with ‘varying columns’ - since a single relation may contain information from different classes of X.500 objects, each tuple may have a different set of attribute types. Although AID does achieve its goal of presenting a more user friendly interface to X.500, it fails to faithfully represent the relational model and SQL, and offers no greater functionality than the X.500 *Search*. Notably absent from AID is the ability to express joins of information from multiple relations. Finally, since AID does not hide the underlying DIT structure, it fails to present a new layer of abstraction, and forces the user to be aware of that structure.

Nomenclator is a QUEL-based interface to X.500. Nomenclator solves the problem of ‘varying columns’ by allowing an administrator to specify which attributes are to belong to a relation. This information, along with other metadata, is stored in a catalog, which the Nomenclator system uses to achieve impressive performance results, while abstracting away from X.500. Like AID, however, Nomenclator allows only a single relation to be involved in each query.

A shortcoming, in our view, of both AID and Nomenclator is their lack of support for queries involving joins. Join queries allow information from multiple relations to be combined, and therefore promote the querying of the relationships that may exist between the modelled objects. We consider join queries to be an important aspect of an SQL interface.

## 2.3 Motivation for the Work - Multidatabase Systems

Our interest in providing an SQL interface to the X.500 Directory Service is the result of research into catalog management for a multidatabase system. Multidatabase systems (MDBSs) provide applications with integrated access to a collection of heterogeneous, autonomous and distributed databases. MDBSs, like conventional database management systems (DBMSs) require that catalog information be maintained in order to provide their services. This information must be stored in a separate repository since component databases are assumed to be independent of the MDBS.

The CORDS MDBS [2] stores its catalog information in a repository based on an X.500 Directory Service. We found that the Directory Service has a number of properties which make it a viable basis for such a repository [4]:

- The Directory Service uses an object-based model which can represent the entities and relationships present in the catalog information.
- The Directory Service supports a variety of attribute types and supports user-defined attribute types.
- The Directory Service provides name resolution for the objects in the MDBS.
- The Directory Service protocols provide a basis for querying and browsing the catalog information.
- The Directory Service provides support for distributed and replicated data.

Two concerns with this approach, however, are the limitations of the query facilities, which are considered here, and the performance and scalability of the Directory Service for this kind of application [9].

## 3 SQL Interface to X.500

The structure of the SQL Interface software is shown in Figure 1. SQL queries to the directory service are parsed and transformed into conventional query trees where the internal nodes of the tree correspond to relational algebra operators (select, project and join are currently supported) and the leaves correspond to "relations" in the DIT. The *metadata* used by the *Query Processor* to generate the query tree and to map between relations and DIT entries is maintained in the *Interface Catalog*. The Query Processor issues requests to a DUA to retrieve data from the directory and performs the relational operations on the data to produce the result.

### 3.1 Mapping from X.500 Model to Relational Model

The basis for our SQL interface to X.500 is the mapping from the X.500 information model to the relational model. This mapping, which allows data stored in X.500 to be viewed as sets of relations

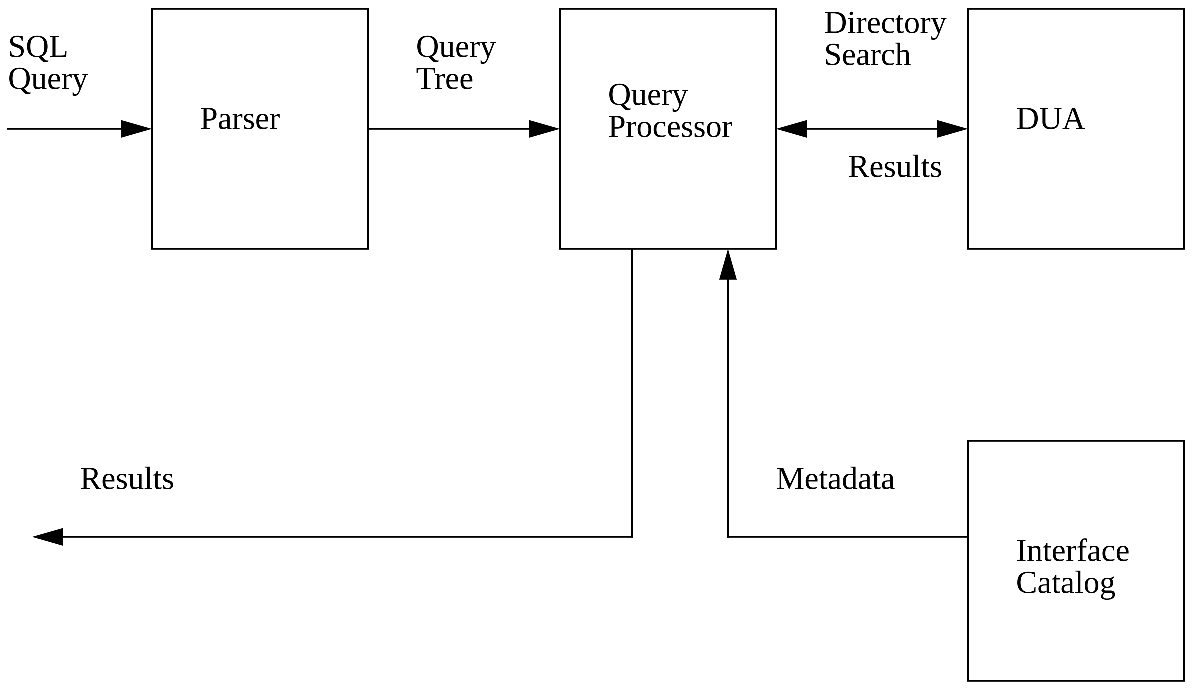


Figure 1: Structure of the SQL Interface

composed of tuples containing attributes, employs the following rules:

- X.500 attributes are mapped to relational attributes. Multivalued X.500 attributes require special treatment, as described below.
- X.500 objects are mapped to relational tuples. We term such objects, *tuple objects*.
- X.500 object classes are mapped to relations. The object class that corresponds to a particular relation is called the *Relation Object Class* (ROC) of that relation. In addition to its ROC, a base object in the DIT is required to uniquely identify a relation. Each relation has such a base object, termed its *Relation Base Object* (RBO). Figure 2 illustrates these concepts.
- The portion of the DN of a tuple object below the RBO is termed its *Effective Distinguished Name* (EDN). Unless otherwise specified and enforced, the attributes that form the EDN of a class of tuple object are selected as the primary key of the corresponding tuples.

First Normal Form in the relational model requires that all attributes have a single, atomic value. X.500, on the other hand, allows multivalued attributes. Aliases even allow a single object to have multiple values for its RDN. In mapping from X.500 to the relational model, there seems to be no

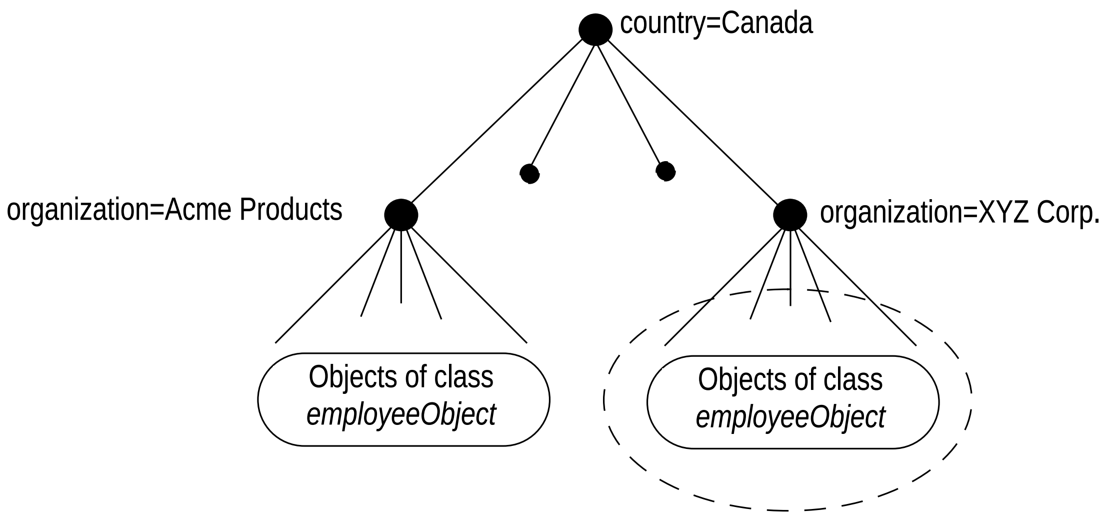

country=Canada
organization=Acme Products
organization=XYZ Corp.
Objects of class *employeeObject*
Objects of class *employeeObject*

Figure 2: Depiction of an ROC and two different RBOs: In a relation in which the ROC was employeeObject and the RBO was 'country=Canada', all objects of class employeeObject in this diagram would qualify for the relation. In this case, the EDNs would include the 'organization' attribute. If, however, the RBO was 'country=Canada@organization=XYZCorp.', only the circled objects would qualify, and their EDNs would be composed of only their RDNs.

reasonable way to preserve First Normal Form. The standard method of creating a new relation for each multivalued attribute would be highly inefficient, since all attributes are allowed to have multiple values (and would therefore require separate relations). Furthermore, there is no obvious way to preserve the relationship between two RDNs that identify a common object.

The solution chosen is to relax the definition of the relational model. Multivalued attributes are allowed in our implementation; multiple values for a single attribute are represented by a comma separated list of values surrounded by braces.[1] Making this change to the traditional relational model, however, introduces a new problem: querying multivalued attributes. For the sake of simplicity, unless otherwise specified, an equality comparison upon an attribute will be interpreted in the multivalued case as an inclusion comparison (i.e. if any of the values of an attribute are equal to the value specified in a query, the given tuple is accepted). Additionally, a strict equality comparison may also be implemented.

[1]This method of dealing with multivalued attributes applies equally to both previously existing X.500 directories and new directories designed specifically for use with this interface. However, it is strongly recommended that any databases belonging to the latter group avoid allowing attributes with multiple values.

### 3.2 Query Processing

A two-phase algorithm is used to process the query trees. The first phase performs a post-order, depth-first search of the tree to perform the necessary X.500 accesses and to expose each operation node to the formats of the tuples upon which it will operate. The second phase of the algorithm involves a series of tree traversals which take the results of the X.500 searches and pass them up the tree.

When a *table* node is visited during the the first phase an X.500 DAP search is executed to retrieve the tuple-objects from the directory which correspond to the table. The RBO and ROC for the table are obtained from the Interface Catalog and the X.500 search retrieves objects of the class ROC in the DIT subtree below the RBO. A filter to restrict the result is constructed from *select* and *project* nodes which immediately precede the *table* node, if they exist.

The result of a search is an array of strings. The array is converted into a collection of tuples: attribute values are placed in the tuple structures, missing attributes in the X.500 objects are converted to the relational NULL value, and multiple values for an attribute are left as a string and placed within braces. The result is stored in main memory as a list which is accessed each time the associated table node is revisited during the second phase of the algorithm. Simple implementations of the select, project and join operations are provided. Details of the algorithm are given in Barrowman [3].

## 4 Performance Analysis

The two objectives of the performance analysis are to gain insights into the feasibility of an SQL interface to X.500 and to point out the most beneficial areas for optimizations. Two test databases are used: a previously existing, "traditional" X.500 directory, and a custom built, "synthetic" relational database. The first two subsections describe the contents and structure of these databases, the queries used to analyze the performance of our X.500 interface, and the results of this analysis. The third subsection compares the performance of the X.500 directory service with a relational DBMS for the "synthetic" database.

All experiments were run on a lightly-loaded system set up as follows:

- The SQL Interface was located on an IBM RS/6000 Model 220 running AIX.

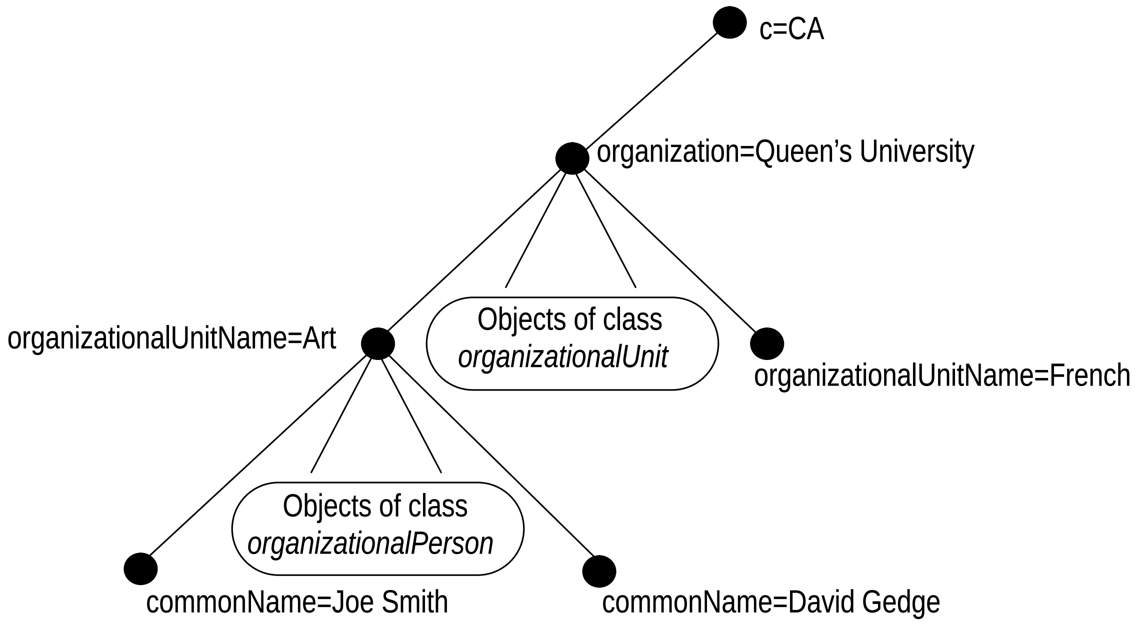

c=CA
organization=Queen's University
organizationalUnitName=Art
Objects of class *organizationalUnit*
organizationalUnitName=French
Objects of class *organizationalPerson*
commonName=Joe Smith
commonName=David Gedge

Figure 3: University White Pages DIT

- The QUIPU X.500 Directory Service was configured with a single DSA running on a second RS/6000 Model 220.
- All indexing and caching was turned off in the Directory Service.
- The performance of the system is presented in terms of the mean time required to complete a query and the mean time required search the directory and to retrieve the necessary data. The time for the X.500 search includes processing times at the DUA and DSA and the communication time. Each query was run a sufficient number of times such that the mean times could be stated with an accuracy of +/- 5% and a confidence level of 95%.

## 4.1 University White Pages Directory

The Queen's University White Pages stores information about the people and departments associated with the university. The structure of the University DIT is illustrated in Figure 3. The root of the university DIT is the 'c=CA@organization=Queen's University' object. Below the root, each member department is represented by an **organizationalUnit** object. Below each **organizationalUnit** object is the collection of **organizationalPerson** objects representing the members of that department. Aliases are used to relate a single person to multiple departments.

Objects of class **organizationalUnit** have a single attribute, **organizationalUnitName**, which is, by necessity, their RDN. Each of these objects in the White Pages DIT has two values for this attribute: a numeric identification number and the name of the corresponding university department. Objects of class **organizationalPerson** have the attributes **commonName** (which is the RDN for these objects), **surname**, **mail**, **telephoneNumber**, and **roomNumber**. Additionally, each **organizationalPerson** object inherits the value(s) of the **organizationalUnitName** attribute(s) of the **organizationalUnit**(s) with which it is associated. We map the X.500 schema to a relational schema with two relations:

- **Departments** with the multivalued attribute **organizationalUnitName** and
- **People** with attributes **commonName**, **surname**, **mail**, **telephoneNumber**, **roomNumber**, and **organizationalUnitName**.

The set of queries used in the experiments are shown in Appendix A and represent the "typical use" of the directory. The queries were derived from data available in a set of log files from the Queen's University public X.500 server. We found that users commonly search for information about a person using attributes such as the surname or common name and perhaps the organizational unit, that is, the department. Thus the set of typical SQL queries only involve selections.

White Pages Queries 1 through 5 are translated into single X.500 search operations which select the final results directly from the directory. Since we wished to verify the assumption that queries which *push down* the selection criteria into the X.500 search are more efficient than those which include the selection higher up in the query tree, we restated White Pages Queries 1 and 3, labelled 1-NF and 3-NF[2] respectively, so that they were implemented by retrieving the entire relation from the directory and then filtering out the unwanted tuples.

Table 1 presents the performance results obtained for the White Pages Queries. For each query, the size of the result relation, the mean time required to process the query, and the percentage of time spent performing the X.500 search is given. We observe the following about the results:

- Queries 1-NF and 3-NF exhibit very poor performance compared to Queries 1 and 3:
    - they take approximately 50 and 27 times longer, respectively, to perform the X.500 search;

[2]'NF' stands for "No Filter" to indicate that no filter is applied during the X.500 search.

| Query | Size of Result (tuples) | Total Time (s) | X.500 Search (s) | % Time in X.500 Search |
|---|---|---|---|---|
| 1 | 1 | 1.56 | 1.50 | 96.3 |
| 2 | 1.54 | 1.54 | 1.48 | 96.2 |
| 3 | 35.78 | 3.01 | 2.54 | 84.4 |
| 4 | 1 | 1.56 | 1.51 | 96.3 |
| 5 | 1.05 | 1.62 | 1.56 | 96.5 |
| 1-NF | 1 | 80.61 | 67.84 | 84.2 |
| 3-NF | 35.78 | 82.08 | 69.71 | 84.9 |

Table 1: Performance of White Pages Queries

- they take approximately 8 and 27 times longer, respectively, to perform the processing after the search;
- they require more memory, since all of the tuples in the **People** relation must be brought into memory.

This confirms our assumption that, for optimal performance, any selection operations in the query tree should take place during, rather than after, the X.500 searches.

- Queries 1, 2, 4, and 5 all require approximately the same amount of time to be processed (between 1.54 and 1.61 seconds), despite the fact that Queries 4 and 5 include the **organizationalUnitName** (part of the EDN of the tuple-objects) in their selection criteria. Additionally, each of these queries requires approximately the same amount of time to perform the necessary X.500 search. This implies that the Directory Service is not taking advantage of the extra information in the query when performing the searches for Queries 4 and 5. It is assumed that the same holds true for Query 3.
- The majority (between 84 and 97 percent) of the total time required to process each of these queries is spent performing the X.500 search operation.

## 4.2 Synthetic Database

Bitton, et al.[5] argue that in order to effectively benchmark a database system, the testbed database must be large, well structured, and composed of uniformly distributed, random "synthetic" data which facilitates the design of queries which retrieve specific amounts of data. The benefits of

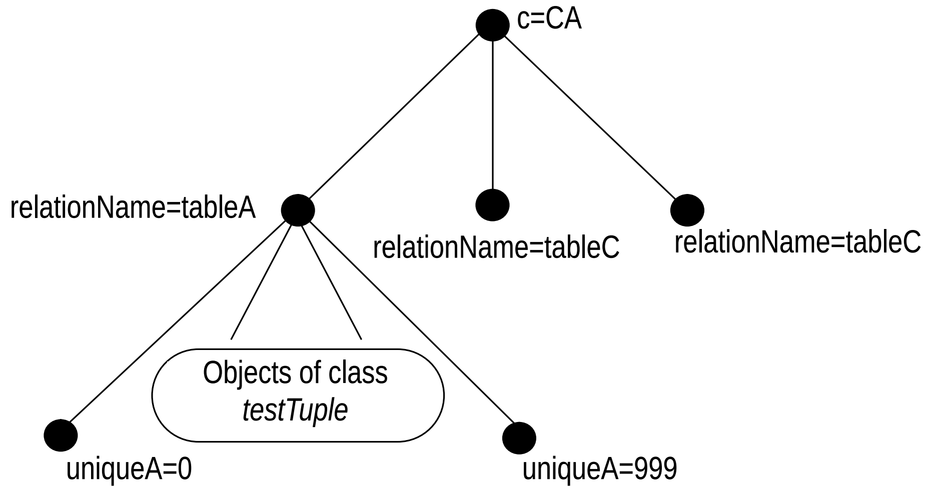


Figure 4: Structure of the Synthetic Relational DIT

structuring the test database in this fashion are that queries are easy to design and understand, and the results of the queries can be precisely controlled.

Our Synthetic Database, which is based on the Wisconsin Benchmark Database, is composed of three relations: **tableA**, **tableB**, and **tableC**. The first two tables contain one thousand tuples and the third contains five hundred tuples.[3] Each of these relations has the same set of attributes: **uniqueA, uniqueB, hundred, ten, four**, and **two**. Attributes **uniqueA** and **uniqueB** take on unique values for each tuple in the relation and, in general, for a given tuple, **uniqueA** $\neq$ **uniqueB**. The remaining attributes are named to indicate the size of the domain from which they take their values. For example, the domain of the attribute **ten** is [0, 9]. The attribute values for each tuple are generated using a pseudo random number generator and are uniformly distributed across the corresponding domain.

The relations belonging to the Synthetic Database are stored in an X.500 DIT as shown in Figure 4. The DIT is rooted at the object "c=CA" which has children of class **relation** that have a single attribute, **relationName**. Our DIT has three objects of this class which serve as the RBOs of the three relations in the database. Since each of these relations has the same attributes, tuple objects under the relation objects all have the same ROC, that is object class **testTuple**. The attribute **uniqueA** is the RDN and the EDN of the tuple objects and the primary key of the tuples in each relation.

[3]Table sizes were restricted by system limitations.

| Query | Size of Result (tuples) | Total Time (s) | X.500 Search (s) | % Time in X.500 Search |
|---|---|---|---|---|
| 1 | 1 | 0.52 | 0.48 | 91.0 |
| 2 | 100 | 5.45 | 4.90 | 89.8 |
| 3 | 10 | 0.98 | 0.91 | 92.2 |
| 4 | 2.5 | 1.0659560 | 0.99 | 93.2 |

Table 2: Performance of Select Queries

| Query | Size of Result (tuples) | Total Time (s) | X.500 Search (s) | % Time in X.500 Search |
|---|---|---|---|---|
| 1 | 100 | 25.00 | 20.38 | 81.5 |
| 2 | 10 | 24.47 | 20.63 | 84.3 |
| 3 | 1000 | 25.38 | 20.10 | 79.2 |
| 4 | 1000 | 50.20 | 44.65 | 89.0 |

Table 3: Performance of Project Queries

The SQL and relational algebra query tree formulations of the queries chosen for the Synthetic Database are presented in Appendix B. We stress queries involving selections on *ranges* of attribute values, projections of attributes, and joins of information from multiple relations, since queries of these types are typical of relational systems, and atypical of traditional X.500 systems. Join queries are of the greatest interest, because that is the relational operation which is least supported by the X.500 information model and DAP, and has not been implemented by other high level X.500 query interfaces.

Tables 2, 3, and 4 present the performance results for our benchmark queries. A number of observations regarding these results can be made:

- For each benchmark query, the system spends the majority of its time performing the necessary

| Query | Size of Result (tuples) | Total Time (s) | X.500 Search (s) | % Time in X.500 Search |
|---|---|---|---|---|
| 1 | 1000 | 110.45 | 89.04 | 80.6 |
| 2 | 100 | 55.72 | 50.59 | 90.8 |
| 3 | 100 | 37.23 | 34.15 | 91.7 |
| 4 | 100 | 10.80 | 9.58 | 88.7 |

Table 4: Performance of Join Queries

X.500 searches.

- Queries that perform a greater amount of filtering during, rather than following, the X.500 search phase demonstrate better performance. This is illustrated by the processing times of the Join Queries 1, 2 and 4. Selections are added to one and then both branches of Join Query 1 in queries 2 and 4, respectively. The processing times drop significantly each time a selection is added.

- The performance of Project Queries 1, 2 and 3 are approximately the same because in each case 1000 single attribute tuples are retrieved from the directory. Duplicate elimination is performed by the query processor. The increase in time required for Project Query 4 is due to the increase in the number of attributes returned per tuple. The extra time is spent in the X.500 system.

- Queries involving join or project operations exhibit particularly poor performance results, since they require entire relations to be retrieved from the directory.

## 4.3 Comparison with Relational DBMS

The above experiments indicate the absolute performance of the X.500 directory system for the given sets of SQL queries. We repeated the Synthetic Database experiments using the Empress[4] relational DBMS in order to judge the relative performance of the X.500 implementation.

The experiments were run on a lightly-loaded system set up as follows:

- The SQL Interface and the DBMS were located on a single IBM RS/6000 Model 220 running AIX.

- The database was queried using SQL through an Open Database Connectivity (ODBC) interface.

- Each of the relations was indexed on the **uniqueA** attribute.

A comparison of the X.500 and DBMS processing times is presented in Table 5. The X.500-based system, as one would expect, does considerably worse than the DBMS for the project and join

[4]Empress is a trademark of Empress Software Inc.

| Query | Empress System Processing Time (s) | X.500 System Processing Time(s) |
|---|---|---|
| Select 1 | 2.24 | 0.52 |
| Select 2 | 2.73 | 5.45 |
| Select 3 | 6.28 | 0.98 |
| Select 4 | 6.48 | 1.07 |
| Project 1 | 7.24 | 25.00 |
| Project 2 | 6.66 | 24.47 |
| Project 3 | 9.43 | 25.38 |
| Project 4 | 10.54 | 50.20 |
| Join 1 | 21.17 | 110.45 |
| Join 2 | 16.37 | 55.71 |
| Join 3 | 7.70 | 37.23 |
| Join 4 | 5.35 | 10.80 |

Table 5: X.500 versus Relational DBMS

queries. While some of the difference may be accounted for by the fact that the X.500-based system must go across the network, the main reason is that the X.500 directory service is not designed to handle project or join internally. The data must be moved out of the directory and the operation performed by the SQL interface. The X.500-based system does, however, perform better than the DBMS on three of the four select queries.

## 5 Query Optimizations

Our observations indicate that the query processing performance of our system is based to a large extent on the performance of the directory service. An obvious direction, therefore, for optimizing queries is to improve the performance of the directory service. Two possible ways to do this for a given query are to reduce the amount of data retrieved from the directory and to reduce the size of the portion of the DIT searched.

### 5.1 Query Rewriting

One approach to reducing the amount of data retrieved for a given query is to rewrite the query tree, so that, while its result relation is unchanged, its directory accesses call for less data. One optimization of this type, which was discussed earlier is to push all select and project operations, where possible, down to the X.500 search. The extreme case of this optimization is illustrated by

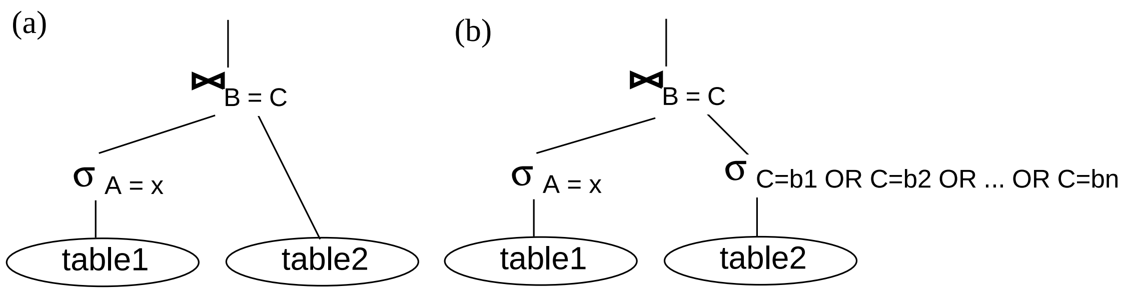


Figure 5: Optimizing a join by introducing a new select operation when the join attribute is not the same as the previously present select attribute: (a) the original query tree; and (b) the optimized query tree.

| Selectivity (%) | Time to Retrieve All Tuples in tableA (s) | Time to Process Select Query 5 (s) | Percentage Improvement (%) |
|---|---|---|---|
| 0.1 | 53.17 | 0.51 | 99.0 |
| 1 | 53.17 | 1.19 | 97.8 |
| 10 | 53.17 | 8.20 | 84.6 |
| 20 | 53.17 | 16.03 | 69.9 |
| 30 | 53.17 | 24.94 | 53.1 |

Table 6: Performance of Select Query 5 for Different Selectivity Factors

the results of White Pages Queries 1 and 3 versus their '-NF' counterparts (as shown in Table 1).

A second type of query rewrite applies to join queries. If one branch of the query tree below the join includes a selection operation, a *join index* [6] may be established. This technique should be applied to the less restrictive branch of the query tree. Metadata describing the size of relations may be required to determine which branch is more restrictive. In the specific case where the selection condition in the more restrictive branch is of the form '$A\theta x$' (where $\theta$ is a comparison operator) and the join condition is of the form 'A=B', it suffices to simply insert a selection with the condition '$B\theta x$' into the other branch. Join Query 4 uses this technique to optimize Join Query 2, and a comparison of the results of these queries in Table 4 illustrates the benefits of the optimization.

In the more general case, as shown in Figure 5 (a), where the selection condition is of the form '$A\theta x$' and the join condition is of the form 'C=B', a join index must be created dynamically. Thus, when the tuple objects that satisfy the selection '$A\theta x$' are retrieved from the directory, the values for attribute C are examined. These values are placed in a set $\beta$, which is then used to create a selection condition of the form 'B=b1 OR B=b2 OR ... OR B=bn', which is inserted into the less restrictive branch of the query tree, as shown in Figure 5 (b).

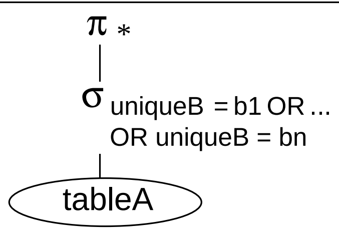
| Query | SQL Formulation | Relational Algebra Query Tree |
|---|---|---|
| Select Query 5 | `SELECT *`<br>`FROM   tableA`<br>`WHERE  uniqueB = b1`<br>`   OR  uniqueB = b2`<br>`   OR  ...`<br>`   OR  uniqueB = bn` |  |

Figure 6: SQL and Query Tree Representations of Select Query 5

The effects of this type of optimization are shown by reformulating Select Query 5 for the Synthetic Database as shown in Figure 6. The results of executing the reformulated query for varying selectivity (that is, with a varying number of elements in $\beta$) are presented in Table 6. The query simulates the branch of the query tree below the join which is being optimized. As we can see, this optimization is extremely beneficial for $\beta$ containing up to 300 elements, or 30% selectivity. Unfortunately, due to system limitations, the technique could not be tested for $\beta$ containing greater than 312 elements. However, if the almost linear trend continues, the use of a join index would be advantageous even for $\beta$ as large as 600 elements.

## 5.2 Pruning The DIT

We observed during analysis that, when querying the White Pages Database, the system was unable to take advantage of the structure of the DIT. This was illustrated by the fact that White Pages Queries 1, 2, 4, and 5 all took approximately the same amount of time to process. We now examine how DIT pruning influences the performance of the system when accessing each of the test databases.

### 5.2.1 Pruning the White Pages DIT

As shown in Figure 3, the level of child nodes directly below the RBO of the White Pages DIT partitions the tuple objects into subtrees based on **organizationalUnitName**. In effect, these nodes act as an "index level" in the DIT, since they can be used to navigate directly to groups of objects which all have the same **organizationalUnitName**. The system should be able to use this fact to achieve improved performance for queries which select tuples with a specific **organizationalUnit-Name** value.

| Query | With No Optimization (s) | With Subtree pruning (s) | Percentage Improvement (%) |
|---|---|---|---|
| 3 | 3.01 | 1.45 | 51.7 |
| 4 | 1.56 | 0.34 | 78.1 |
| 5 | 1.62 | 0.36 | 77.5 |

Table 7: Performance of White Pages Queries 3, 4, and 5 with Subtree Pruning

The system may restrict its search of the DIT by augmenting the RBO passed to the X.500 search with the additional information regarding the **organizationalUnit** to which all result tuples of the query must belong. This allows the search to focus on the only subtree of the DIT which may possibly contain valid result tuple objects, thereby minimizing the number of objects examined. When a query is submitted to the system, the select filter being passed to X.500 may be examined. If it contains a reference to a specific value of the index attribute, the DIT may be pruned.

Table 7 compares the performance of White Pages Queries 3, 4, and 5 when no optimization is used and when the DIT pruning technique is used. It is clear that the optimization is beneficial - performance is improved by more than 50 % for Query 3 and by more than 75 % for Queries 4 and 5.

### 5.2.2 Pruning the Synthetic DIT

The simple structure originally chosen for our Synthetic DIT does not allow for any form of DIT pruning, beyond the relation level. In order to take advantage of pruning, a new level of "index objects" must be introduced which sits between the RBO and the tuple objects of each relation, as shown in Figure 7. The index level is composed of objects of class **indexObject**, which have a single attribute, **hundred**. For each value of the **hundred** attribute, there is a corresponding instance of **indexObject** below which all tuple objects with that value reside.

As with the White Pages Database, using DIT pruning with the Synthetic Database involves examining a query for a reference to the **hundred** attribute in its selection condition. If such a reference exists (as it does, for example, in Select Queries 3 and 4), the relevant **indexObject**, rather than the RBO of the relation, is chosen as the base object for the X.500 directory search. This allows the system to focus on a much smaller subset of tuple objects (in our case, 100 times smaller, on average). However, for queries which cannot take advantage of the index objects, the DIT

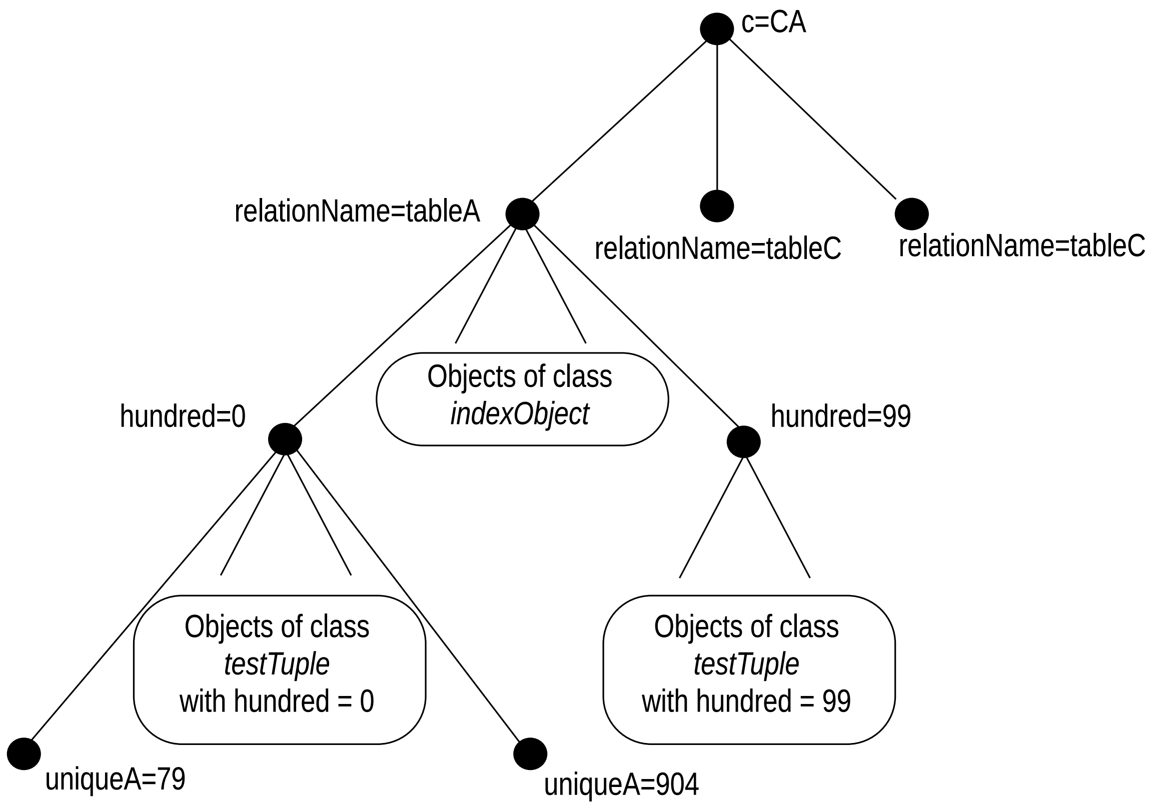

c=CA
relationName=tableA
relationName=tableC
relationName=tableC
Objects of class *indexObject*
hundred=0
hundred=99
Objects of class *testTuple* with hundred = 0
Objects of class *testTuple* with hundred = 99
uniqueA=79
uniqueA=904

Figure 7: The Synthetic DIT with objects of class indexObject introduced for optimization purposes.

| Query | With No Optimization (s) | With Subtree pruning (s) | Percentage Improvement (%) |
|---|---|---|---|
| 1 | 0.52 | 0.63 | -21.0 |
| 2 | 5.45 | 6.21 | -13.9 |
| 3 | 0.98 | 0.92 | 5.9 |
| 4 | 0.83 | 0.59 | 28.7 |

Table 8: Performance of Select Queries with Subtree Pruning

is actually larger and more complex than before, due to the additional level of objects. These facts are reflected by the results presented in Table 8. Select Queries 3 and 4, which can take advantage of pruning demonstrate improved performance, while Select Queries 1 and 2 exhibit an increase in response time.

A variation of DIT pruning, which we call *DIT pruning with aliases*, allows pruning to be implemented without reducing the performance of queries that do not use it. For each relation, the original DIT subtree is left unchanged and a second subtree is created which has at its root an "index RBO" for the relation. The children of this object are the **indexObjects** described above. The children of these objects are *X.500 aliases* to the tuple objects. Thus queries which cannot use the index objects follow the original DIT and queries which can use the pruning method select the appropriate **indexObject** as the base object for the search, and then follow aliases to the tuple objects.

Unfortunately, experiments demonstrated that while the performance of queries which do not use the pruning was unaffected, as predicted, the performance of queries using pruning was significantly worse than with plain DIT pruning. It appears that the additional time required to resolve aliases in our DIT is considerably greater than the time required to simply examine each subtree below the index level.

Our results therefore indicate a rule-of-thumb for implementing DIT pruning:

> If an index level class already exists (as it does in the White Pages DIT), pruning should always be beneficial.
>
> If, however, we are free to choose the structure of the DIT (as is the case for our Synthetic Database), an index level class should only be included if it is known that queries which take advantage of pruning will dominate those that do not.

## 6 Conclusions

The X.500 Directory Service, while initially intended as a white pages service for communication entities in a distributed system, has properties which make it a candidate for storing other types of data needed in a distributed system. Foremost of these properties is the directory's ability to handle replicated and distributed data. Two potential problems with using the directory service as a general database system are its limited interface and its questionable performance in this role. The work presented here addresses the first of the problems by describing an SQL interface to an X.500 Directory Service.

Two aspects of the SQL interface are described: the mapping between the relational model and the X.500 information model, and the query processing strategy. The mapping gives a faithful representation of the relational model,which can be applied both to previously existing X.500 directories and to DITs developed specifically for use with this interface. It allows a single DIT to contain multiple relations which can be referenced unambiguously, thus permitting queries involving joins to be expressed. Furthermore, it presents a new layer of abstraction on top of the X.500 information model, allowing the underlying DIT structure to be hidden. Our use of the interface with a University White Pages directory demonstrates that SQL provides a convenient mechanism for formulating queries to the directory service.

A performance analysis of two sets of SQL queries is presented. One set of queries is for a traditional White Pages Database and the other is for a relational-style Synthetic Database derived from the Wisconsin Benchmark Database. Performance of select queries is acceptable and, in some cases, better than that of a commercial relational DBMS. Performance of project and join queries using the X.500 Directory Service is worse than with the DBMS in all cases. This deficiency can be attributed to the fact that project and join are not "natural" queries for a directory service.

An examination of the performance of the queries indicated that most of the time processing a query was spent in the X.500 Directory Service which suggested that the focus of optimization should be on reducing that time. We proposed and evaluated two optimization techniques. The first optimization technique is intended for join queries. It is based on the notion of a join index and it rewrites the query to include an exhaustive select of tuples from one branch based on values obtained in the other branch. This resulted in improvements in the time to process the one branch from 50 % to 99 % depending upon the selectivity of the join index. The second optimization technique, DIT pruning, reduces the amount of the DIT that must be searched for a query by exploiting knowledge of values in a parent node of the DIT. This technique can not be used in all cases but, applicable queries showed improvements from 5 % to 77 %.

Further analysis of the performance and scalability of the X.500 Directory Service must be performed before we can conclude whether or not the Directory Service is a viable candidate for a general database in a distributed environment. The results presented here are encouraging and the Directory Service may be prove to be a reasonable approach for certain types of data.

## Acknowledgements

The authors would like to thank Wendy Powley and Lu Li for their help in this project.

## References

[1] S. Abrutyn. AID: SQL Interface to QUIPU. Technical Report 90-5, Center for Information Technology Integration, University of Michigan, 1990.

[2] G. Attaluri, D. Bradshaw, N. Coburn, P.-Å. Larson, P. Martin, A. Silbershatz, J. Slonim, and Q. Zhu. The CORDS multidatabase project. *IBM Systems Journal*, 34(1):39–62, January 1995.

[3] D. Barrowman. An sql-based interface to x.500. Master's thesis, Department of Computing and Information Science, Queen's University at Kingston, March 1995.

[4] M. Bauer, N. Coburn, P.-Å. Larson, and P. Martin. Managing global information in the CORDS multidatabase system. In *Proc. of Second International Conference on Cooperative Information Systems (CoopIS'94)*, pages 23–34, Toronto, May 1994.

[5] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking Database Systems - a Systematic Approach. Technical Report 526, University of Wisconsin - Madison, 1983.

[6] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin Cummings, Redwood City, CA, 1989.

[7] J. Hong, M. Bauer, and J. Bennett. Integration of the directory service in distributed systems management. In *Proceedings of the 1992 International Conference on Parallel and Distributed Systems*, pages 142–149, 1992.

[8] S. Kille. The Design of QUIPU. Technical report, Department of Computer Science, University College, London, 1988.

[9] P. Martin and W. Powley. Storing MDBS Catalog Information in an X.500 Directory Service. In *Proceedings of the 1994 CAS Conference*, pages 216–226, 1994.

[10] J. J. Ordille and B. P. Miller. Nomenclator Descriptive Query Optimization for Large X.500 environments. In *Proceedings of the 1991 SIGCOMM Conference*, 1991.

[11] C. J. Robbins and S. E. Kille. *The ISO Development Environment: User's Manual - Volume 5: QUIPU*, 1994.

## A SQL Queries for University White Pages Database

## B SQL Queries for Synthetic Database

## C

## D