Using X.500 to Facilitate the Creation of Information Systems Federations

by

Eric Kolotyluk
B.Sc. University of British Columbia 1981

THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
December 1994

Approval

Name: Eric Kolotyluk

Degree: Master of Science

Title of Thesis: Using X.500 to Facilitate the Creation of Information Systems Federations

Examining Committee

Chair: Lou Hafer

Jia-Wei Han
Senior Supervisor
Associate Professor of Computing Science

Peter Triantafillou
Supervisor
Assistant Professor of Computing Science

Tiko Kameda
External Examiner
Professor of Computing Science

December 9, 1994
date approved

Abstract

X.500 is the international standard for a world-wide automated directory system, the Directory, which enables people and automated systems to search for information such as people, places, systems, services, etc. However, much of the information that is expected to be found in the directory already exists in corporate and institutional databases as well as other information sources.

In the last ten years or so, computing experts have begun creating federations of information systems in order to give us easier access to a variety of information sources. These range from simple but powerful network grazers like Archie and Veronica to more sophisticated heterogeneous database projects like Interbase, Myriad, ORECOM and others.

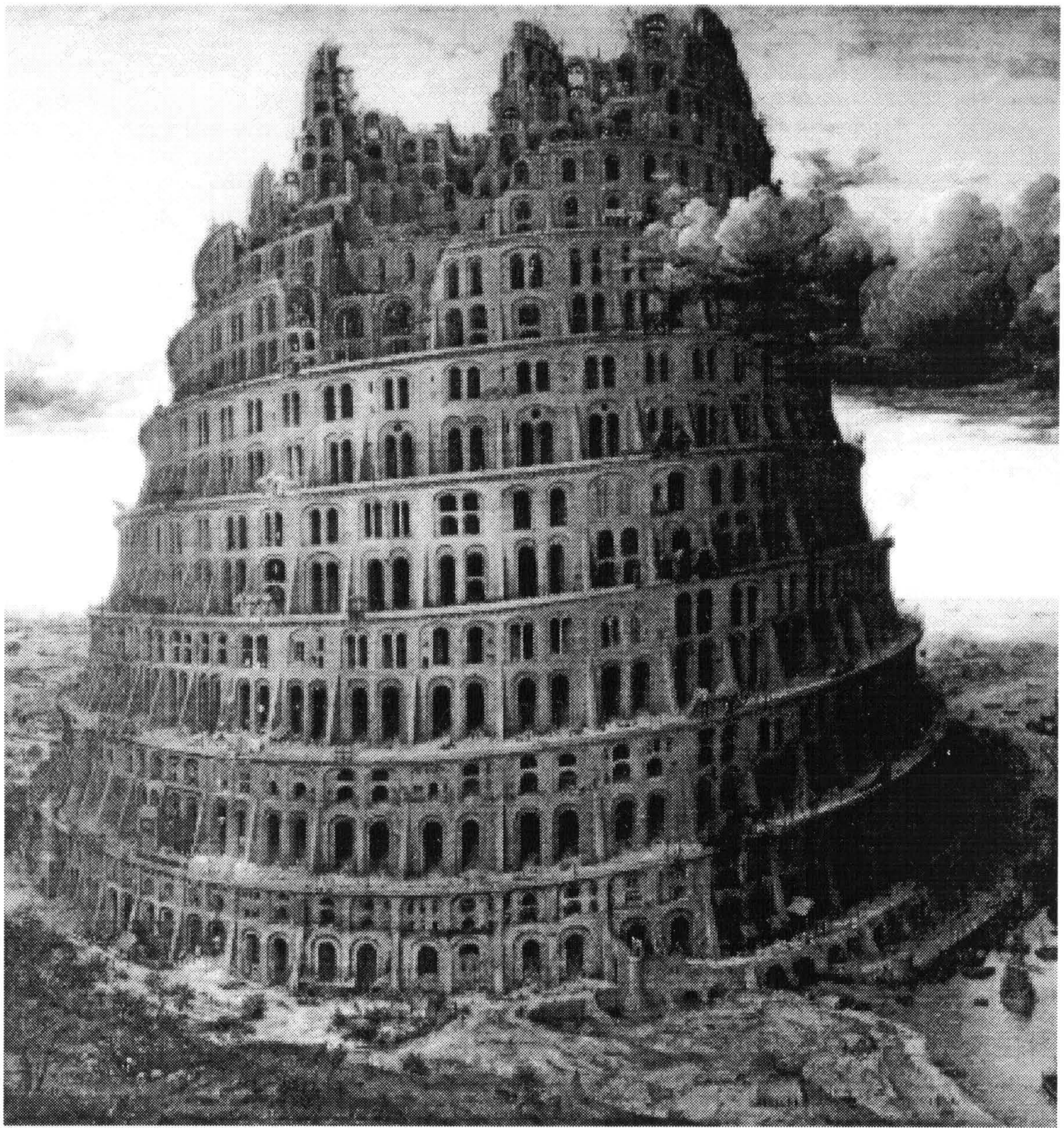
The issues of interfacing the X.500 Directory with existing information sources are explored—in effect, the X.500 Directory is seen as a federation manager of information systems. In particular, the integration of X.500 methods with relational database technology is studied in the context of what is possible and what makes sense. To test these ideas an existing Directory Service Agent has been modified to access a commercial relational database management system, allowing a directory administrator to map the tuples of a relation into the entries of the directory. Results from this effort have revealed challenges not only in the conceptual design of such a system, such as information schema translation, but also in the pragmatics of system design, such as the registration of external information sources within the Directory itself.

Finally, this thesis speculates on the ultimate scope of X.500 as an information systems federation manager, where the global information network could be headed and what we might see or invent along the way.

Dedication

to Jane Durrant

for lasting inspiration



Quotation

And the whole earth was of one language, and of one speech.

And it came to pass, as they journeyed from the east, that they found a plain in the land of Shinar; and they dwelt there.

And they said one to another, Go to, let us make brick, and burn them thoroughly. And they had brick for stone, and slime they had for mortar.

And they said, Go to, let us build a city and a tower, whose top *may reach* unto heaven; and let us make a name, lest we be scattered abroad upon the face of the whole earth.

An the Lord came down to see the city and the tower, which the children of men builded.

And the Lord said, Behold, the people is one, and they have all one language; and this they begin to do: and now nothing will be restrained from them, which they have imagined to do.

Go to, let us go down, and there confound their language, that they may not understand one another's speech.

So the Lord scattered them abroad from thence upon the face of all the earth: and they left off to build the city.

Therefore is the name of it called Babel; because the Lord did there confound the language of all the earth: and from thence did the Lord scatter them abroad upon the face of all the earth.

Genesis, Chapter 11, verses 1-9

Acknowledgments

X.500 has enormous potential as a guide through the ever expanding morass of information and world-wide information systems. The many people who contributed to its design should take pride in their work. Without the grassroots development of a public domain X.500 service at University College London there would not be the hundreds of X.500 sites throughout the world. In particular Steve Kille should be applauded for his vision and contributions to the effort. Allan and Tim Howes should also be recognized for their technical support at the end of e-mail wire (as well as their patience with at least one frustrated developer). Finally, without the initiative and exceptionally hard work of Marshall Rose, there would not have been an ISODE, upon which the majority of X.500 sites are based.

The original topic of this thesis was going to be something like "benchmarking commercial database management systems for use with X.500." Credit should be given to my supervisor Peter Triantafillou who said this topic was not interesting enough and proposed the broader, 'more interesting' topic on which this thesis is based.

The statement "may you live in interesting times" was said to have been a curse, and like much interesting work this thesis and its related project were not without their curses. Fortunately, in addition to his academic expertise, the continuous support, patience and motivation of my senior supervisor, Jia-Wei Han, made these curses more bearable and kept the work going.

Table of Contents

Chapter 1 Introduction

In the early to middle 1980's information management technology began creating heterogeneous information systems in order to give us easier access to a variety of information sources. This can be seen with the commercial success of information gateways like Envoy-100 in Canada; Dialog, CompuServe, the Source, and EasyNet in the U.S; and Inspect, Questel and 'ii' in Europe. More recently in the 1990's the popularity of 'free' information browser/finder applications like Archie, Veronica, Gopher and Mosaic has built upon the success of the Internet and has changed the style of information access by acting more as information brokers than as gateways. However, even these services still do not tap into more sophisticated database systems because of the more sophisticated information models. To address this research, efforts like Interbase [27], Myriad [36], ORECOM [32], IDEF-1X [32], EXPRESS [32], NIAM [32], and OSAM* [32] are attempting to create powerful schema definition and translation methods which will give us the capability for information retrieval from a diversity of information management systems.

The reason these efforts are so popular is that progressively our society depends on information as a resource. As more information is collected into mountains, it increasingly becomes important to be able to mine for useful information efficiently. The main problem is that often each mountain of information requires a distinct method of information mining. Thus, there are too many distinct methods and users rarely have the time to learn all theses methods or have the resources to acquire them. What people increasingly desire is a common means of accessing a wide variety of information sources. Heterogeneous information systems promise the ability to hide several distinct information mining (access) methods behind such a common interface.

As an endorsement of this new heterogeneous information technology, many of the reference papers used in this Thesis were obtained in a short time browsing the World-Wide-Web via NCSA's X/Mosaic application, instead of the many hours or days more traditional library searches and other means would generally require. However, it still takes many hours and days to read the rapidly increasing research results in this budding area.

1.1 The Directory

X.500 [1,2,Appendix A] is the international standard for a world-wide automated directory system[1] which enables users (people and automated systems) to search for information about people, places, and other things. From the user's perspective, the Directory is similar to services like the Internet Name Service, and the PH servers for POP mail, but vastly richer in features and sometimes equally frightening in complexity. In most cases it is typically used for looking up electronic mail (e-mail) addresses of people, but there are many efforts to expand the range of uses.

A related effort by the Object Management Group (OMG) is to create the Common Object Request Broker Architecture (CORBA) [5] so that client applications may interoperate with objects on other systems. An important part of this is the ability to search for objects and control access to objects through an object directory mechanism. Indeed the CORBA directory design attempts to take into account existing directories such as X.500.

In the abstract sense the Directory is much more than a means of looking up e-mail addresses. It is foremost a standard for Directory User Agents (DUAs) to communicate with Directory Service Agents (DSAs). The DUA is the user interface to the Directory (which is a collection of co-operating DSAs). The DSAs share the management of persistent information known as the Directory Information Base (DIB). What is interesting and important in this case is that X.500 in no way specifies how the information is stored in the DIB, but how it is accessed and managed. In this sense the DSA is like an SQL (Structured Query Language) program, in that it only specifies how to access the information but the actual physical storage and management of the information is an implementation detail separated from the use of an SQL program. In fact, while SQL has been primarily used with Relational Database Systems (RDBS) there have been attempts to use SQL with other types of database systems.

[1] Generally called the OSI Directory, or just 'the Directory.'

1.1.1 Other Directories

Directories are vital in an information-dependent culture such as ours. Imagine going to a library with hundreds of thousands of books but no catalogue, or trying to find a plumber to fix your sink but not having a telephone book or directory assistance.

On any computer system, the first operation *everyone* learns to do is search for files in the file system directory. Without the file system directory it is hard to imagine how we could ever make effective use of computers. However, the file system directory is not the only directory of important information. In the ever popular Unix, for example, the files `/etc/passwd`, `/etc/group`, `/etc/hosts`, `/etc/networks`, and so on, are all directories for various types of information. One unfortunate consequence of this implementation of directories is that each has its own particular syntax and access methods. A program (or person) that is looking for files in the file systems has to use one method, but when looking for host computers on the network, has to use a very different method.

In an environment where myriad computer systems and networks are the norm, network-wide directories become increasingly important. In particular, there becomes a need to base directories not in the context of a single computer system, but the network as a whole. Sun Microsystems' Network Information Service (NIS) is an example of an attempt to rationalize the directory in terms of the network. However, NIS only provides a minor abstraction on the collections of file-based directories, and does not integrate the file system information with other information. More importantly, the original NIS only supported the concept of a network, and not a hierarchy of networks as is often the administrative reality.

Efforts were made in the early to middle 1980's to develop a more generalized distributed directory system with the Internet Name Service (or just name service). This defined a truly hierarchical and slightly more abstract service in which people and automated systems could query the name service for information about named entities. While the name service is a vital service in the international Internet, it was only designed to address an immediate need to look-up network addresses of host computer systems and was extended slightly to do directory-based routing of electronic mail addresses. While other uses were proposed for the name service, it is rarely used for more than host address lookup and mail routing.

Other popular directory systems include usenet (network news), Gopher, Archie, and Mosaic. These are all light-weight standards in that they are designed to be easy to implement and do not require a great deal of rigor in defining information standards, especially in terms of schema. Each of these systems is also very specifically designed for a particular application, often requiring different access methods for very similar information.

There is an increase in building gateways from one information service to another and in integrating information bases. For example, there is now a common Gopher/ Mosaic information server that permits administrators to combine the information base resulting in reduced storage requirements as well as improved information synchronization. There are even Gopher to X.500 and Mosaic to X.500 gateways permitting Gopher and Mosaic users to browse information in the international X.500 directory.

1.1.2 Why X.500?

While development of X.500 started around the same time as the Internet Name Service, the design goals of the international standards community were significantly less ad-hoc and substantially more ambitious in terms of strictness of standards and richness of features. In fact, the feature that distinguishes X.500 from all the previously mentioned directory technologies and standards is that each of the previously mentioned directory techniques is an ad-hoc attempt to design and implement a service. While such ad-hoc methods have produced impressive results, none has really attempted to produce a generalized information directory service that is truly multifunctional.

X.500, on the other hand, specifies a very generalized directory standard for looking up information on arbitrary types of objects. While the Internet Name Service, Gopher and Mosaic are capable of presenting very general types of information, they are much weaker in their ability to search for specific types of information. For example, in the INS you can search for a name of a unique object, but you cannot search for all the objects of a particular type of host other than listing all hosts and doing the search yourself. In

Mosaic (and the World-Wide Web) you can search for information based on keywords, but only if someone has created an keyword/index server for a particular subject area. Furthermore, because schema standards are so much less formal, it makes it more difficult for automated tools to search and utilize much of the valuable knowledge available. For example, while Mosaic is a very successful hypertext/hypermedia browsing utility, the 'hyper-links' are typically built by hand or by custom ad-hoc indexing methods. Having a standard form of entity attributes like X.500 would facilitate automatic creation of such hyperlinks.

Table 1: X.500 Features

feature	description
hierarchical	Like the INS, Gopher, and Mosaic, X.500 facilitates easy browsing of information structures.
naming	Any particular object modeled in the directory can be uniquely identified with a well defined hierarchical naming structure. While the Internet Name Service (INS) supports a simple hierarchical name structure, names carry far less information. For example: INS: eric@sfu.ca X.500: country=CA@organization=Simon Fraser University@organizationalUnit=Faculty of Applied Sciences@organizational-Unit=School of Computing Science@commonName=Eric Kolotyluk While the X.500 name provides far more information about the named object, it is not practical to remember or type, and so 'User Friendly Names' can be used, specifying partial information. UFN: Eric Kolotyluk, Simon Fraser University, CA Unlike Gopher, Mosaic offers its own naming system, http names, which can locate a 'page' of information via a variety of access methods. This leads to cumbersome, difficult-to-remember names with a complex syntax. While X.500 names can seem long and cumbersome, the syntax is simpler as any information on access methods is removed from the name and can be found in the attributes of the entry the name represents.
schema control	All information attributes must be encoded in a well specified manner. This includes the names of attributes, the syntax of each type of attribute value, the required and allowable attributes of each class of objects represented in the directory, the allowable object classes allowed at particular locations in the directory. This can make it easier for automated systems to make sense of directory information since it is more standardized.

Table 1: X.500 Features

feature	description
information update	Unlike services such as the INS, Gopher, and Mosaic, it is possible for the user to update information in the directory through the same mechanism for accessing the information. In these services it is necessary for a systems administrator to update such information. Part of the reason for this is that, unlike X.500 with its well established schema controls, it can be very dangerous for inexperienced users to update the information bases in these other services.
authentication	X.500 supports the notion of a directory user 'binding' to the directory, potentially requiring that the user authenticate themselves through passwords or public key techniques. This is even more important when people may have update access to information.
access controls	Some implementations of the Directory support access controls on information in the directory. In Quipu, Access Control Lists (ACLs) can specify who is allowed to read, write, and compare a particular directory entry (object), the child entries of that entry, or even individual attributes of the entry. This has very important implications in information access legislation (rights to access and rights to privacy) that have not begun to be explored in other information services.
search mechanisms	While not as powerful as mechanisms like SQL, X.500 provides powerful search mechanisms that enable directory users to search for specific entries (objects) under a wide range of conditions.
information modeling	With the growing research and development in knowledge-based intelligent systems, it is increasingly important that the information be available in a well-defined formal way. X.500 provides a very logical and object-oriented model of information.

One particular ideal of the author is that from the computer users' perspective, at sometime in the future we should be able to forgo the great diversity of directory and information access techniques, and utilize a common, powerful mechanism, a super browser or automated librarian to direct us to and access all the available information resources on the international computer networks. X.500 offers more of the necessary technical infrastructure to make this possible than systems like the Unix file system, Gopher and the World-Wide-Web.

1.2 Information Reuse and Synchronization

While X.500 may appear to some as an elegant information management system, it is just another information management system. Much of the information that is expected to be found in the directory already exists in corporate and institutional databases and other information sources. In order to make use of this information it must be translated from its native format to a format that is suitable for use in the Directory. In such a situation, there are really only the following two approaches handling multiple information access mechanisms:

1.2.1 Replication and Static Translation

In this approach, the information from one system is replicated and translated to another. This is generally the most straightforward approach for a number of reasons:

1. Once the information is translated it can be accessed immediately in the new system.
2. There are generally a great many tools to aid in static translation. For example, on Unix: ed, awk, pearl, and several other tools exist to transform information in batches.
3. Hand tuning of the translation is generally quite practical because this translation is often a once only exercise.
4. Most vendors provide tools to translate external information sources into their own format to encourage users to continue using their product.

A significant drawback to this, however, is that it is possible for the two systems to disagree, over time, on the correct information (diminishing information integrity). If someone updates information about a person on the first system, but is unaware of the replicated information on the other system, the second system will no longer have consistent replicated information.

In order to keep replicated information consistent it is necessary to devise and implement synchronization mechanisms. Typically this involves either periodically comparing the information in the two systems and resolving inconsistencies (yet another interesting problem), or updating both information systems when ever a change is made to either. In this last case we have effectively Replication with Dynamic Translation.

Eventually if replicated information is to be kept consistent, there will always have to be mechanism for keeping the information consistent. In the long run this will always require at least as much effort as implementing pure Dynamic Translation.

1.2.2 Dynamic Translation

Rather than replicating the information from one system, the information is translated dynamically each time a request is made for it at the second system. In essence, this second system acts as a gateway to the first system. The main advantage of this approach is that inconsistency of replicated information is impossible because there is no replicated information. Another advantage is that as there is no replicated data, extra storage space is not required.

Another variation is to allow replication via caching. This can enable better performance of information access and increased availability of information, but also leads to many problems of synchronization and information integrity. In many cases, lazy propagation of new information to the cache would be tolerable. For example, on an hourly or daily basis the directory could refresh its cache.

The primary disadvantage to Dynamic Translation is that it can be quite a lot of work up front to devise and implement effective automated mechanisms to translate information between two or more systems. With replication and static translation it is possible to get a new directory system in service more quickly, postponing issues of information consistency until later. Another disadvantage of Dynamic Translation is that if the primary information system malfunctions the information will also not be available to the directory.

Issues of replication and caching in distributed-replicated file systems have been explored in systems like Coda[6], Sprite[7,8], and Locus[9].

1.3 The X.500/Relational Connection

There are at least three basic reasons why someone would want to connect the Directory to a relational database management system:

1. To provide the persistent storage for the Directory Information Base (DIB).
2. To provide a means for Directory users to view the low-level raw information.
3. To allow Directory users higher-level access to an existing database.

While each of these goals has its own set of characteristic challenges, they are also related in that there are good reasons to combine any two or all three for added benefits. In terms of classifying the issues, however, this is a good division of approaches.

1.3.1 Using a Relational Database System for the DIB

Above all, one of the most important features of the Directory is that it has a persistent store for information. On top of that it is desirable that the Directory has safe and efficient access to that information. While some X.500 implementations have chosen to create their own persistent store, others have used existing database management tools to implement the DIB.

The main advantages to implementing a customized persistent store are that:

1. It can be more expedient. For example, the initial implementation of Quipu used a simple, human readable/editable ASCII text file called an Entry Data Block (EDB) file for each level of the Directory Information Tree (DIT). When the Quipu DSA is started, all the EDB files are loaded into main memory and only used again for update.
2. The persistent store can be customized and tuned to the specific requirements of the Directory.
3. Dependence on proprietary systems can be avoided. For example, in a public domain[1] system like Quipu, accessibility and usability of the overall system can be greatly compromised by dependence on proprietary components.

[1] While Quipu and ISODE are not exactly in the 'public domain,' the availability of and distribution of these systems are very similar to public domain systems.

On the other hand, there can be some disadvantages to creating a customized persistent store:

1. Expedience often means trade-offs. For example, in Quipu: the DSA start-up time can be very large (an hour or more), the virtual memory demands can be difficult to manage[1] (a hundred megabytes or more), and directory updates can be quite slow[2] (many second).
2. Customization often means 'reinventing the wheel.' Generally this means implementing custom mechanisms for data integrity, data indexing, back-up and recovery, transaction management, etc.
3. Dependence on non-proprietary or public domain technology can be limited in terms of available technology and ongoing support.

The main advantages to using existing tools and products to implement a persistent store are:

1. It can also be more expedient to avoid 'reimplementing the wheel.'
2. Using existing mature technology can take advantage of previous effort and expertise at performance tuning, data reliability, and data management facilities.
3. Opportunities to integrate and interoperate directory information with information in existing databases.

The disadvantages to using existing tools and products, however, include:

1. Selection of an existing database management System typically requires some expertise in that area. For example, appropriate use of an Relational Database Systems (RDBS) may require substantial expertise on the part of the directory developer in architecting an appropriate database for a DIB.
2. An impedance mismatch between needs of the DSA and the access methods of

[1] Excessive VM usage can lead to excessive paging, resulting in more time spent in paging input/output than other useful work. Quipu paging can interfere with other applications and visa versa, requiring the systems administrator to isolate Quipu and move other applications to different systems.

[2] Slow individual updates can make bulk update of directory entries prohibitive. This typically requires locking the DIB from directory updates, updating the DIB through other means, then restoring normal operation of the Directory.

the DBMS can lead to problems of performance, complexity, maintainability, etc. For example, Sybase provides character string data-types for up to 255 characters, but X.500 routinely requires attribute values of more than 255 characters.

3. Reliance on proprietary technology can limit access to the particular directory implementation because of licencing restrictions, royalties, government restrictions[1], etc., as well as increase the fiscal dependence on commercial products.

Ultimately, the goals and needs of the directory developer will determine a given choice of DIB implementation. In some cases a combination of methods can be appropriate. For example, in the current version of Quipu's disk-based DIB (via GDA, the Generic Directory API) the Gnu Database Management (GDBM) library is used to implement the persistent store and indices for the DIB, while allowing great flexibility in customizing the persistent store to the needs of the Quipu DSA.

In regards to using a RDBS for implementing the DIB, the specific advantages to doing this are:

1. Access to mature technology. The leading RDBS vendors offer products built on many years of research, development, implementation experience, and user testing. In fact, there is probably more known about relational database technology than any other database management technology.
2. Access to existing mechanisms important to DIB operation includes: performance enhancements such as indexing mechanisms, data integrity, and transaction management[2]. Furthermore, contemporary RDBS products such as Sybase support multiprocessor host systems.
3. Many organizations, especially large ones, that are likely to run a directory are

[1] For example, the US government is very proactive in limiting technology for export to other countries, especially in the area of security technology such as encryption (an area very important to the OSI Directory).

[2] For example, the ISODE Consortium plans to implement DSA performance increases by making the existing DSA software 'thread safe' and adding a multiprocessing package. While this may be generally worthwhile, existing database management systems already support such performance features. By contrast, an alternate to adding multiprocessing to ISODE code would be to run multiple DSAs of the same DIB and rely on the RDBS for managing concurrent access to the DIB.

also likely to have a commercial RDBS product. Having a DSA that utilizes their existing RDBS would let them exploit existing expertise and investments in that technology. In particular if that organization already has database information that it would like to incorporate/interoperate with the directory, it would generally be much easier to construct mechanisms to move data from existing databases to the DIB database, and keep each database in synchronization with the other.

4. Relational databases offer a great deal of flexibility in database management that would allow a DSA administrator to fine tune the DIB to organizational requirements. For example, in most RDBS products:
   a. Indices can be created and dropped at anytime. Also, there is typically the option to create different kinds of indices. This would give the DSA administrator great flexibility in experimenting and performance tuning of the DIB while it is already operational.
   b. SQL can allow extremely powerful access to the DIB. In particular, SQL can offer the DSA administrator (and in some cases the Directory user) much more powerful access to the DIB than through the standard X.500 search and update mechanisms.
   c. Views can be utilized to implement a degree of schema interface independence between the needs of the DSA and the actual DIB implementation.
   d. Many RDBS products have optimizers that can compile access plans for views and SQL procedures. These are transparent to the operation of the DSA and views and procedures can be recompiled whenever the underlying DIB changes (i.e., index changes). Also, some RDBMS products (e.g., Sybase) offer hand-tuning of the optimized plans that would give the DSA administrator even more ability to fine tune the DIB to organizational needs.

In spite of the attractive features RDBS technology might offer to the DSA/DIB developer, there can also be disadvantages:

1. While not necessarily a limitation of the relational model, contemporary products have significant limitations in the data-types they provide. For example, in

Sybase the data-type 'varchar' would be a popular choice for storing X.500 attribute values. However, the maximum length of a varchar is limited to 255 characters. Although Sybase provides the 'text' data-type for character strings longer than 255, this data-type cannot be accessed via SQL in the same manner as varchar. Furthermore, for every 'text' value, a minium of 2048 bytes of storage is allocated in the database—consequently, text strings only slightly longer than 255 characters result in very poor storage utilization.

2. SQL was designed primarily for interactive database access by a person. While many implementations provide program access through SQL, more programming effort is required to access the data than systems like ObjectStore where the primary interface is via persistent variable in C^{++} (see below).
3. Associative access to data can incur a performance penalty. Even with efficient index mechanisms like B-trees and hash tables, joining tables via indirect keys imposes far more overhead than resolving direct pointers.
4. Inventing a relational schema appropriate for Directory use can be a challenge. Because relational systems offer such great flexibility in database schema design a great deal of expertise and experience is required to design an effective schema. Furthermore, the issues raised in the previous points make this design task even more challenging.

Using an OODB for a DIB

While relational database technology may be a very attractive platform for a Directory Information Base, Object-Oriented Database Systems (OODBS) can offer features not always found in relational technology, which can, however, also be of benefit to a DSA/DIB implementor. Because the X.500 model includes object-oriented features, the notion of class and subclass for example, there may be closer schematic relationships between the X.500 model and a given OODBS.

However, semantic kinship is not necessarily the most attractive feature of an OODBS. Recent research and development in OODBS design has led to a much greater variety of features between OODBSs than between contemporary RDBS products so it is difficult to explore this kinship with every significant OODBS.

ObjectStore from Objective Designs Inc. belongs to a class of OODBS designs known as the 'persistent store.' Previous work by the author investigating the application of ObjectStore as a DIB showed:

1. The persistent store model offers a better impedance match between the persistent data, data structures, and DSA code. This allows easier DIB implementation compared to that possible with more conventional RDBS technology which typically requires access through SQL and cumbersome data transfer mechanisms.
2. Greater flexibility in data types and structures also makes it easier to meet DIB requirements. For example, ObjectStore data types are those of C++, while Sybase data-types are much more limited in use. More specifically, a 250-byte character string is treated the same as a 260-byte character string in ObjectStore, whereas the same cannot be handled effectively in Sybase.
3. ObjectStore's 'collections' class provides powerful tools (B-trees and hash tables) for creating custom index methods directly. This also makes it possible to efficiently implement associative search with performance rivaling a RDBS, in addition to the highly efficient direct access characteristic of OODB methods.

Interestingly enough, the object-oriented nature of ObjectStore was of little value with respect to paralleling the O-O schema of X.500. Rather than statically implementing the X.500 schema, ObjectStore was used to create a general mechanism where the schema could be maintained dynamically. In this respect the same can be done via an RDBS.

Implementing a DIB via ObjectStore also has its drawbacks. Some of the disadvantages include:

1. Schema evolution is still more difficult in ObjectStore than Sybase. While improvements are being made in this area, at the time ObjectStore was investigated for DIB design and implementation, there were no schema evolution mechanisms.
2. While ObjectStore does support powerful associative search capabilities, these must be implemented directly into DIB code and optimization is left up to the DIB implementor. Relational systems typically support effective optimizers and the flexibility to tune performance after the database has been implemented.

Research has shown that it can be difficult to match Sybase's associative search performance using ObjectStore.

3. ObjectStore has no SQL interface. SQL can be a very powerful tool for a directory administrator to analyze directory information or perform information maintenance.

In spite of some serious evolutionary shortcomings, technology such as ObjectStore would likely make for a more effective DSA/DIB implementation than a RDBS. This is true, however, only if there will be no interoperability between the DIB and existing relational databases.

There is still much room for improvement in RDBS implementations, especially in the area of impedance match with programming languages. If products like Sybase were able to support the entire C++ data type and data structure suite the way ObjectStore does, it would be a lot more desirable to use a RDBS for a DSA/DIB implementation. Similarly, if better interfaces between program code and the underlying RDBS engine could be developed, then RDBS-based DIB implementation would be much easier.

1.3.2 Using the Directory to View Raw Relations

Another intriguing use of the Directory would be to directly access the relations of a given RDBMS. In a sense this would be quite straightforward, for example, given a relation of products:

Table 2: Example Parts Relation

number	name	price	stock	description
12345	HD107-1	$215	239	107 MB SCSI-1 hard disk
12346	HD223-2	$398	465	223 MB SCSI-2 hard disk
12347	HD457-2f	$765	310	457 MB fast SCSI-2 hard disk
12348	HD768-2f	$1133	127	768 MB fast SCSI-2 hard disk
12349	HD1024-2f16	$1370	97	1024 MB 16-bit fast SCSI-2 hard disk

The directory administrator might like to map this relation into a directory subtree with the following entries:

```
organizationalUnit=Stock Parts
                   ├──────────── commonName=HD107-1
                   ├──────────── commonName=HD223-2
                   ├──────────── commonName=HD457-2f
                   ├──────────── commonName=HD768-2f
                   └──────────── commonName=HD1024-2f16
```

Performing a directory read on commonName=HD768-2f might be expected to return something like:

```
commonName=HD768-2f,
                partNumber=12348,
                price=$1,133,
                stockQuantity=127,
                description=768 MB fast SCSI-2 hard disk
```

It might also seem reasonable to assure that, under the appropriate circumstances, a directory user be able to update this database via directory access methods. For example, the parts/inventory manager might want to change the description to "1024 MB wide fast SCSI-2 hard disk." If the manager was already viewing the inventory via the X.500 user interface, it would be easier to update the information via X.500 than to start up an SQL user interface and reformulate the update in SQL.

Overall this example demonstrates a useful purpose for viewing raw relations and is fairly straightforward to implement. However, there are a number of issues that would have to be addressed.

Database Schema

Rarely are tables in relational databases as simple as Table 2. More often than not, a relation will have a number of fields which are (direct) foreign keys, or related attributes will be found in other relations linked by their (indirect) foreign keys. Finally, the likelihood of dealing with direct and indirect foreign keys typically increases for normalized databases.

The straightforward solution to the problem of dealing with indirection via foreign keys is to use relational views. By creating a view relation that joins all the necessary tables into an appropriate view and eliminates foreign keys which would not be meaningful if viewed through the directory, there again is a more straightforward mapping of the relation into a set of directory entries. The fundamental catch to this solution is the classic "view update problem" in the case where someone wanted to update the database via the directory. While some commercial RDBMS products allow updates via views, it is generally under very restrictive conditions. However, in many cases this is an easy limitation to live with, especially if update through the Directory is not required.

Another problem is that views often are joins across other relations. When this happens the resulting information in the relation is no longer normalized, for example, Table 11, "PhoneUsers," on page 34 is the result of joining the tables Person and Telephone. One undesirable artifact is that the person, "William Havens," appears twice in the view but we would not want two entries for "William Havens" to appear in the Directory.

These issues will be explored further in section 1.3.3 "Integrating the Directory With Existing Databases" on page 20.

Directory Schema

We must face the Directory schema with its extensive control mechanisms [1,2]. In particular, schema controls in the Directory would typically require the following restrictions on the mapping of the parts table:

1. Each entry in the subtree 'oranizationalUnit=Stock Parts' would also require an objectClass attribute to specify the required and allowable attributes of each entry. This objectClass would have to be previously defined to the DSA in question.
2. Each attribute-type of each part entry would also have to be previously defined to the DSA.
3. Each attribute-type would also have to have an associated attribute-value syntax defined.

While many would consider this a simple matter of configuration, such configuration is typically tedious and error-prone and would be better achieved through automatic methods. Certainly there is enough information present in the definition of the parts relation that the appropriate schema could automatically be created in the Directory. Unfortunately the current state of X.500 schema management does not yet allow this.

Schema Differences

Aside from the stringent schema enforced by the Directory there are some differences in the Directory's schema and that of the basic RDBMS.

1. The Directory schema allows multiple values for a given attribute. Contemporary RDBSs do not yet deal well with this concept. Typically the tuple is either represented twice, with each distinct value, or the particular attribute is expressed through a foreign key and a one-to-many relationship. There is however theoretical work [10,11,12,13] that simplifies the expression of multi-valued attributes in relational databases.
2. The directory is fundamentally a hierarchy, or tree, of information, hence the term Directory Information Tree (DIT). Consider the following directory:

```
organization=Simon Fraser University
   |
   |--- organizationalUnit=Faculty of Applied Sciences
   |        |
   |        |--- organizationalUnit=School of Computing Science
   |        |        |--- commonName=Eric Kolotyluk, position=staff
   |        |        |--- commonName=Jia-Wei Han, position=faculty
   |        |
   |        |--- organizationalUnit=School of Engineering Science
   |                 |--- commonName=Chao Cheun, position=staff
   |                 |--- commonName=Jim Cavers, position=faculty
   |
   |--- organizationalUnit=Academic Computing Services
            |--- commonName=Lional Tolan, position=staff
            |--- commonName=Robert Urquhart, position=staff
```

Consider a more typical relation of people in university departments where we must find an appropriate mapping from departments to organizationalUnits:

Table 3:

department	position	first name	last name
Computing Science	staff	Eric	Kolotyluk
Computing Science	faculty	Jia-Wei	Han
Engineering Science	staff	Chao	Cheng
Engineering Science	faculty	Jim	Cavers
Computing Services	staff	Lionel	Tolan
Computing Services	staff	Robert	Urquhart

While there may seem an obvious mapping to the reader, the basic problem is: how do we specify this mapping in a more formal way so that a program may perform the mapping automatically?

3. Another significant problem has to do with 'schema content.' Not only may the schemas differ in structure, but in what they contain. For example, in the previous university directory, there are only four attributes in each person entry. But what if a person in the directory also requires a phoneNumber and electronicMailAddress attributes. These attributes are not represented in the relation, so where do they come from? Do we extend the relational schema with the new attributes or do we store these in the directory, and what are the pros and cons of each technique?

Summary

The basic problem with mapping raw relations into X.500 schema is that schema management in the Directory is still not fully mature [4]. While there has been much effort in this area, in particular, the addition of service attributes in the 1988 version of X.500, there are few, if any, implementations of X.500 1988.

In the very near future there will likely be enough support for and experience with schema management in X.500 to make it extremely straightforward to map raw relations into X.500 schema. This would make it commercially attractive for organizations to 'publish' portions of their existing relational databases via the Directory. In this way customers and potential customers could browse databases such as parts, reducing the need for vendor newsletters, advertising, price lists, and other junk mail. Consequently, this would make it commercially attractive to develop DSAs that can easily provide X.500 access to existing relational databases.

1.3.3 Integrating the Directory With Existing Databases

Using a RDBS to implement the DIB, or using the Directory to view raw relations seems to be an acceptable and useful reason for integrating X.500 with a RDBS, but each has certain limitations. In the first case, if we want the DIB to interoperate with an existing information source, we still have to translate the information structures and the schema between the two models. In the second case, we may not be able to capture the full meaning of the entities in the database through one relation or even through a view.

The last significant approach is to access the relational database at a higher conceptual level, namely, the actual database level as opposed to the raw relation level. In this case, rather than forcing the relational database administrator to create a new database for the DIB, or creating the appropriate views to map into the Directory schema, the DSA becomes a user of the actual database.

We saw a hint of this sort of problem in the previous section where we needed to deal with foreign keys in raw relations, in particular, with handling multi-value attributes. The approach was to create a new view of the data which more closely resembled the structure of a directory entry. By extending this technique we would ultimately deal with the entire relational schema by defining a set of views to support each particular type (or object class) of directory entry.

Chapter 2 Heterogeneous Information Systems

In Genesis chapter 11 from the Bible we see an interesting story of people, all of whom speak the same language, seeming to work well enough together that even the Lord was concerned of what they might achieve. He was so concerned that He chose to intervene by *confounding* their language and scattering them abroad. We can at best speculate as to what danger the Lord saw, although one popular interpretation is that the people building the city and tower were so proud and arrogant, boasting they could reach heaven, that the Lord decided to teach them some humility. Whatever deep truth the story holds, one fact that we can be sure of is that thousands of years ago people had a *sense* of the power of standards and conventions, or conformity, and were equally aware of the frustrations due to differences in communication and practices.

Indeed, thousands of years later we still grapple with these fundamental issues of conformity vs. individuality, often with debate of religious passion. Ergo, the young field of computing science is no exception as proponents of standardization continually wrestle, intellectually, with proponents of the "free market of ideas and products" [17].

While much can be said, philosophically, about standardization, experience typically shows that standardization works best with mature technology. The nature of a mature technology is that it need not change much. Any improvements in the technology have likely already been incorporated, or can be incorporated within the framework of the standard. The trick, therefore, is how to realize some of the benefits of standardization in a young and rapidly changing technology.

As we saw in Chapter 1, there is increasing pressure on computing professionals to offer better ways for computer users to access information quickly and easily, whatever the fundamental source. Inevitably, many researchers and developers have accepted the challenge, accepted that there is great diversity in information systems, accepted that there is enormous investment in established systems, and accepted the fact that information seekers want access anyway. The trick these people have discovered is to develop technology that accepts these differences as well.

2.1 Terminology

In surveying some of the recent literature [18,27,29,34,35,36,38], there are a number of terms used almost interchangeably: Heterogeneous Databases, Federated Databases, Multidatabases, and Interoperability. One of the best overviews of this terminology comes from Elmagarmid and Pu's Introduction to the Special Issue on Heterogeneous Databases in the September 1990 issue of the ACM Computing Surveys [17].

2.1.1 Heterogeneous Databases

Basically a Heterogeneous Database Management System (HDBS) is one where two or more distinct information sources are accessed as though there were one homogeneous information source. These sources can be heterogeneous in terms of features like the data model, query language, database schema, transaction processing, etc.

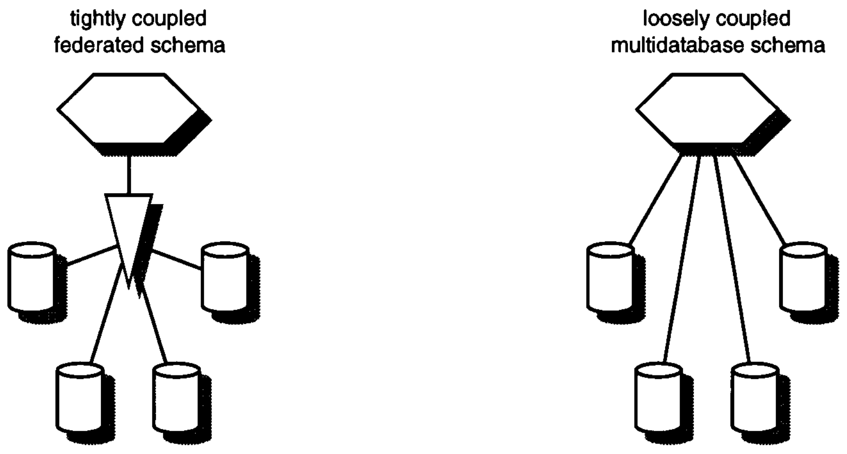
2.1.2 Federated Databases

According to Webster's dictionary, a federation is 'a union of organizations.' To most people the common example of a federation is that of the federal government; a union of the provincial or state governments. Most of us will also be familiar with the distinctive nature of each of these separate organizations and the difficulty keeping the federal process working. In many places in the world, it is nearly impossible to keep any large federation of governments together at all for more than a few years at a time. However, we continue to persevere forming federations and attempting to keep them going because, in spite of all the problems, we find greater value in having them.

The most distinctive characteristic of a Federated DataBase System (FDBS) is the effort to tightly couple the schema of the component databases. In particular, there is quite a bit of effort in designing the 'federation manager' to be extremely flexible in accommodating the various database schemas, typically through a single canonical form which the other schemas are mapped into. From the user's point of view there is really only one schema to consider. The main disadvantage is that it can be very difficult to find a canonical form for databases with wildly different schemas.

2.1.3 Multidatabases

In [17] a multidatabase system (MDBS) is defined as "a collection of loosely coupled element databases, without an attempt to integrate them using a unified schema." In practice, there is a wide range of what we might call multidatabase systems, from information gateways like, Envoy, Dialog, CompuServe, The Source, EasyNet, Inspect, and Questel, to information finders like Archie and Mosaic, to full-fledged DBMSs like Sybase, Empress V2, Ingres/Star, and Oracle V5. In the latter cases, these full-fledged DBMSs have typically been extended with MDBS concepts and already share a common data model, although they may have different schema. In this author's opinion, it is misleading to include Sybase, Empress V2, Ingres/Star, and Oracle V5 in the category of multidatabases.

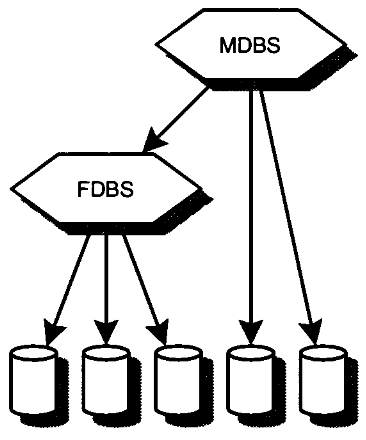


The main disadvantage of a multidatabase system is that the user will generally have to be more aware of the different schemas and other characteristics of the component databases. The advantage of this, though, is that for wildly different schemas the multidatabase manager need not be too ambitious in its design and implementation by relying on the 'intelligence' of the human user. In short, the multidatabase approach is a more pragmatic one, while the federated database approach is a more idealized one.

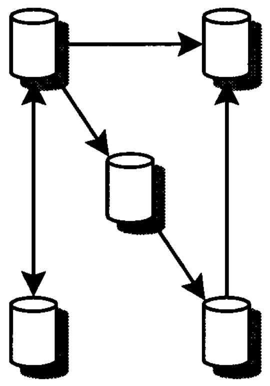
What is important to remember about both of these approaches is the similarity of the results—unified access to a heterogeneous set of database systems. While starting with different assumptions, each approach works towards a similar goal that will become increasingly similar as technology improves.

2.1.4 Interoperability

While the term interoperability is used often in the literature, there are few attempts to define it clearly with respect to federated, and multi-databases. The impression is, though, that interoperability is a horizontal concept, as opposed to a vertical one. That is, in a federated database or multidatabase the component systems are 'driven' by a managing agent, the FDBS or MDBS, and do not interact or interoperate with each other directly. In this sense there is a hierarchy of control. On the other hand, interoperating database systems can interact with each other directly. Rather than a hierarchy of control, there is a lattice of control.



federated systems


interoperating systems

In this sense, the X.500 Directory accessing information in a RDBS is also an issue of interoperability. If we were to implement an SQL interface to X.500, we could realize two-way interoperability between X.500 and relational databases without the need of a FDBS or MDBS. In another sense, we can view both X.500 and the RDBS each as a heterogeneous database system. The basic limitation of not having a FDBS or MDBS component is that each interoperating database (or information system) component is limited to its own data model and operation suite, whereas a FDBS or MDBS can be designed to encompass the data models and operation suites of its constitute members.

There are trends to increase interoperability with efforts like ANSI SQL, SQL2, and SQL3 [15], by adding features which make it easier to support broader information models. Over the long run we should expect to see increasing convergence between the FDBS-MDBS approach and the interoperability approach, to the extent that every leading DBS interoperates with every other DBS and each becomes a Federated DBS or Multi-DBS. This should be a general goal for the information management community as, in the end, it is not so important how we get to the information as how effectively we get the information.

2.1.5 Levels of Heterogeneity

Within the field of heterogeneous information systems there are certain characterizations we can make about so-called systems that we can group into levels of heterogeneity.

In [33] the authors try to characterize levels of heterogeneity in their "Spectrum of Cooperative Problem Solving." In this characterization there are two dimensions of criteria, Collaboration vs. Coordination, which they use to define the following spectrum: Centralized Databases, Distributed Databases, Federated Databases, CSCW, Office Automation, Distributed Artificial Intelligence Problems.

While there are many other possible characterizations of heterogeneous database methods and systems, the important point is that there are different levels or degrees of heterogeneity to be considered.

Level 1 Heterogeneity

This class of heterogeneous systems includes systems where the various data schemas may differ, but where the underlying data management model is the same. For example, this would include a federated database system where the constituent members are all relational database systems, possibly all made by the same vendor.

Another possibility would include a federation of OODBSs [30], but the significant bulk of research is in fact focused on the heterogenous relational database systems. This makes a certain degree of sense in that the bulk of production database systems are relational, yet unable to interoperate with each other.

Publications in this category include [29,30,36,37].

Level 2 Heterogeneity

This class of systems includes systems where the underlying data management models are different. For example, this would include a multidatabase system where the constituent members include distinct RDBS and OODBS products from different vendors.

While there is definitely growing research into systems which exhibit this level of heterogeneity, it is definitely a more difficult problem to address as not only do all the Level 1 challenges have to be addressed, but the additional task of dealing with sometimes profoundly different information management models.

Publications in this category include [22,27,31,32,34,39].

Level 3 Heterogeneity

This class of systems is more difficult to define, but is intended to describe heterogeneous information systems where the constituent members are radically different beyond Level 2. For example, one might envision a heterogeneous information system where the constituent members include relational, object-oriented, geographic, and CAD databases, as well as information sources like on-line documentation, hypertext, multimedia browsers, and spreadsheets. This definition is intended to make a distinction between systems with powerful data models like database management systems, and systems with little or no data models.

Publications in this category include [27,33].

2.1.6 Research Areas

Within the field of heterogeneous database research there are several broad areas of study:

1. Architecture
2. Schema Integration and Translation
3. Transaction Processing (commitment and recovery)
4. Query Processing and Optimization.

Of the four, architecture is probably the most important because it sets the framework for all the other pieces of the system. Architecture involves not only the architecture of the component databases of a HDBS, but the architecture of the HDBS itself. Indeed, when surveying the literature, it is common to find some discussion of architecture even if the core issue is either Schema Integration and Translation or Transaction Processing, Commitment and Recovery.

Schema Integration and Translation is probably the second most important issue researched. Differences in schema are a direct result of working with heterogeneous databases, where Architecture and Transaction Processing are well known issues from the study of homogeneous databases. In this sense Schema Integration and Translation are probably the most important area where further research is needed, whereas issues of architecture are more mature and more in need of refinement and standardization.

While transaction processing issues are also important, as they relate to HDBS management, most work is concerned with the extension and refinement of what has been learned about transaction processing in homogeneous database systems. Two key transaction processing issues which are a direct result of HDBS research are:

1. Distributed Processing. Heterogeneous database systems are almost always distributed. As a result, good algorithms for distributed transaction processing are fundamental to the success of any heterogeneous database system.
2. Autonomous Processing. More germane to HDBS transaction processing is the reality that autonomous and often different models of transaction processing must be accommodated.

Although transaction processing is also quite important to the task of utilizing X.500 as a HDBS, it will not be covered in the rest of this thesis in order to limit the scope and avoid tempting digression. More information on transaction processing in heterogeneous database systems can be found in [26,45,46,47,49].

Like transaction processing, Query Processing and Optimization in an HDBS is really an extension to what has already been learned from homogeneous database systems. Two key issues relevant to this area are:

1. Query Translation. Because more than one component DBS is involved in a HDBS, query translation is unavoidable. This may be case of translating one higher level query into two or more queries (in the same query language) for component DBSs, or it can be a matter of

2. Autonomous Processing. Like transaction processing, the autonomous operation of constituent DBSs can present a problem for query processing, in particular for query optimization.

While query processing, and in particular query optimization, are important issues even in the context of X.500 as a heterogeneous database, the issues have more to do with making database processing work better than work at all. For this reason, and to also limit the scope of this thesis, the subject will not be covered further, except for query translation which can be closely related to schema integration and translation. Further information on query processing and optimization can be found in [24,26].

2.2 Schema Integration and Translation

Fundamentally the most challenging issues of HDBS operation come from dealing with the realities of differences in schema between constituent DBSs. Differences in schema can be small, on the order of lexical differences;

Table 4: Minor Schema Differences

create table Person	create table Person
(name char(32),	(name char(40),
roomNumber smallint	office char(5)
)	)

more moderate, on the order of syntactic differences;

Table 5: Moderate Schema Differences

create table Person (name char(32), roomNumber smallint)	create table Person (firstName char(20), lastName char(20) officeId smallint) create table Office (officeId smallint, buildingId smallint, room char(8)) create table Building (buildingId smallint, buildingName char(24))

or even quite severe, on the order of semantic differences;

Table 6: Major Schema Differences

create table Person
(name char(32),
roomNumber smallint,
	class Person : Object
	{ name: char(32),
	SIN: int }
	class Employee : Person
	{ department: *Department,
	phoneNumber: char(16) }
	class Department : Object
	{ division: *Division,
	name: char(32),
	manager: *Employee}

2.2.1 Common Data Models

Quite a bit of work [29,30,31,32,39] has gone into the definition of common data models to deal with the problem of schema differences. By defining a powerful data model which captures the essential features of all other significant data models, the number of translators can be minimized. This contrasts with the approach of creating translators between every pair of data models where the number of translators grows on the order of the square of the number of data models. In reality the number of data models does not necessarily grow very large and it can often be easier to design a custom translator than a single powerful central model.

Some of the most recent work has taken the approach of defining a powerful Object-Oriented common data model. The advantage of this approach is that it accommodates the more modern O-O schemas as well as the more common relational ones.

2.2.2 Higher-Order Logics

Once a common data model is defined, it is still necessary to define the translation between existing database schemas and the canonical schema. While a variety of techniques have been proposed [29,30,31,32,36,38], the use of logic programming techniques seems quite promising[29]. In particular, logic programming lends itself well to the definition of meta-level information structures and translation rules. When coupled with an object-oriented information model, the logic programming rules can be utilized as *methods*[1] for data translation and access.

2.2.3 Entity Identification

One of the central problems in schema integration and translation is identifying similar or identical entities in different databases with different schemas. Consider the following example from [37]. We have two tables which use different keys to identify their tuples so we must search other attributes to find matching tuples. In this case it seems the name attribute is a good start, but we may also need to compare the street attribute in Relation A with the region attribute of Relation B.

Table 7: Relation A

aId	name	street	cuisine	manager
a1	Village Wok	Wash. Ave.	Chinese	Lim
a2	Ching	Co.B Rd.	Chinese	Hwang
a3	Old Country	Penn. Ave.	American	Slagle
a4	Old Country	HW. 7	American	Tom

1. In object-oriented terminology a 'method' is a procedure or subroutine that is specifically designed to work on the data of a particular class objects. By encapsulating methods with data the relationship between the data and the methods can be better defined and managed.

Table 8: Relation B

bId	name	region	specialty	ratingB
b1	Village Wok	UofM Campus	Hunan	Average
b2	Ching	Roseville	Sechuan	Excellent
b3	Old Country	Downtown	Steak	Good

2.3 Registration

One of the issues that does not seem to be covered well in much of the surveyed literature is that of registration of external data systems. In particular, registration deals with how one information system is made aware of the existence of another, probably different, information system. Is this 'awareness' built into the underlying framework of the given heterogeneous information system, or does there exist extensible mechanisms for describing new types of information structures, schema translation methods, and the actual communications protocols and connection mechanisms?

Most work in the area seems to assume that registration is an implementation detail, or is built into the underlying heterogeneous design. This assumption is quite likely due to the fact that much of the work in heterogeneous database systems centers around heterogeneous relational database where there is a different data model and schema, but a similar interface such as SQL. In particular, with the adoption of ANSI Standard or Open SQL, it is possible for RDBS products from different vendors to interoperate with each other, so the issue of registrations is a simple configuration issue.

However, there is less (although increasing) work in the field addressed to the interoperability of widely divergent database systems or information sources. In these cases registration is generally ad-hoc and built in to the system design.

2.4 Conclusion

Perhaps database research will never build its Tower of Babel. Perhaps the task is too difficult, or it is just not realistic to have one common database approach. This is clearly the case today, yet it is essential that we build bridges between these towers.

Building these bridges are researchers with a keen sense that while there are differences between the various information storage systems, there is enough commonality between the information stored within them that we can concentrate more on the information and less on the systems. One of the 'building tools' which seems quite promising is the use of logic programming techniques to build powerful schema definition and translation mechanisms.

However, building bridges alone is not likely to be enough. Ultimately, we will need to standardize the bridges. For example, what good is a narrow rope suspension bridge to someone with an ox and a wide cart to haul? Standardization is inevitable though to any mature technology and, as can be seen with standardization efforts such as SQL-2 and SQL-3 to add support for OODB technology, progress continues to be made in this area. Fundamentally, though, it will be our understanding of information and information processes which will give us the ability to define good standards. By wrestling with heterogeneous information systems, we can only hope to improve our understanding of information system.

Chapter 3 Resolving X.500/Relational Differences

As we saw in Chapter 1, there are some tempting similarities between the X.500 Directory model and the relational model, but there are also some significant differences, especially with respect to their schema. Three main problem areas are: dealing with multivalued attributes, mapping the directory hierarchy to the relation structure, and dealing with schema content differences. Once all these issues are addressed, we need to have some means for expressing how schema translation is to be performed.

3.1 Mapping Multivalued Attributes

As was discussed on page 18, the X.500 directory supports multi-valued attributes. While most contemporary RDBM systems do not explicitly support multivalued attributes (because relations must be in 1NF), there are extended relational models such as the nested relational ones [10,11,12,13], which support non-atomic attribute values. If we were fortunate enough to be interfacing the directory with such an extended relational model, we could make use of the nest and unnest operations to more easily interface the X.500 model to relational databases.

In practice RDBMSs do handle multivalued attributes either through resembling tuples[1] or through foreign keys expressing one-to-many or many-to-many relationships. For example, consider the relations People(firstName, lastName, idNumber) and Phone(user, number), where user is a foreign key on idNumber in the People relation. We might create a view by joining two relations, People and Phone, resulting in a number of repeated tuples for people who use more than one phone.

[1] By 'resembling tuple' I mean a tuple where all the fields are the same as some other tuple except for one.

Table 9: People

firstName	lastName	idNumber
Eric	Kolotyluk	123456789
William	Havens	234567890
Jia-Wei	Han	345678901

Table 10: Phone

user	number
123456789	291-3014
234567890	291-4973
234567890	291-4623
345678901	291-4411

Table 11: PhoneUsers

firstName	lastName	idNumber	phoneNumber
Eric	Kolotyluk	123456789	291-3014
William	Havens	234567890	291-4973
William	Havens	234567890	291-4623
Jia-Wei	Han	345678901	291-4411

Table 12: Nested Version of PhoneUsers

firstName	lastName	idNumber	phoneNumber
Eric	Kolotyluk	123456789	291-3014
William	Havens	234567890	{291-4973, 291-4623}
Jia-Wei	Han	345678901	291-4411

One key question is what do we want to model in order to map our relational database into X.500 attribute values?

1. We may choose to map resembling tuples into multi-values. For example, given a view PhoneUsers(firstName, lastName, idNumber, phoneNumber) with tuples {(William, Havens,..., 291-4973), (William, Havens,..., 291-4623)} we could use the SQL statement,

   select phoneNumber from PhoneUsers where idNumber=234567890

   to extract the phone numbers for William Havens. One advantage of this technique is that the relation does not have to be in 3NF[1] [10,11], which is likely in the case of views resulting from a join of two or more relations. The basic disadvantage here, however, is that we quickly run into the view-update problem [10,11], where we might want to add an additional phoneNumber for Jia-Wei Han by way of the view PhoneUsers.

[1] In this case, the view PhoneUsers is not in 3NF because the attribute phoneNumber is not dependent on the primary key idNumber since there is more than one phoneNumber value for 234567890. In this case, idNumber cannot be a primary.

2. We may also choose to make more explicit use of foreign keys. For example, we could just as easily use the SQL statement:
`select number from Telephone, where idNumber=234567890`
to extract the phone numbers for William Havens. The main advantage of this technique is that we don't have to deal with the view update problem as we can simply add another phone number for Jia-Wei Han with something like:
`insert Phone value(345678901, 291-1234)`. The main disadvantage is that our X.500-RDBS gateway now needs to know more about the schema of the underlying relational database. This adds to the overall complexity of the system.

3. In the case of a nested RDBS, we might make use of SQL-2 or SQL-3, which are new versions of the SQL standard permitting multivalued attributes.

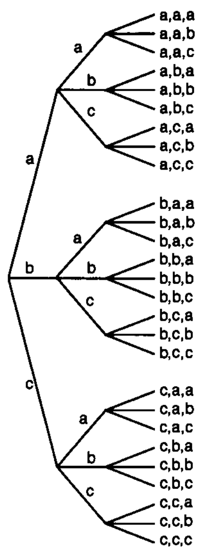
In practical terms, we should support both techniques 1 and 2 as it is entirely possible that a given database may not be in 3NF, but we still want to interface with it.

3.2 Mapping between Directory Hierarchy and Relation Tuples

In the discussion on the Directory's hierarchical schema in Section 1.3.2 the problem of mapping this hierarchy to the structure of a relation was pointed out. The following discussion explores this issue more deeply.

There is a tempting duality between tree hierarchy and relational tables that we might use to map between the structure of the hierarchy and the information in the tables. In particular, for any relation R with attributes k_1, k_2,... k_n that form a candidate key, one can represent this as a tree hierarchy where k_1 is the first level of the tree, k_2 is the second level, and so on. Conversely, given an arbitrary tree, one can construct a relation by creating a candidate key with attributes k_1, k_2,... k_n where n is the number of levels in the tree. In Figure 1, consider a relation where three attributes make up a candidate key, where each attribute has three distinct values, and where every possible combination of unique primary keys exists. In this case k_1 corresponds to the first level of the tree, k_2 to the second, and k_3 to the last level.

Figure 1. Duality of Candidate Key Attributes and Tree Hierarchy



k_1	k_2	k_3	other attributes
a	a	a	
a	a	b	
a	a	c	
a	b	a	
a	b	b	
a	b	c	
a	c	a	
a	c	b	
a	c	c	
b	a	a	
b	a	b	
b	a	c	
b	b	a	
b	b	b	
b	b	c	
b	c	a	
b	c	b	
b	c	c	
c	a	a	
c	a	b	
c	a	c	
c	b	a	
c	b	b	
c	b	c	
c	c	a	
c	c	b	
c	c	c	

First we consider the case where we want to map a given Directory hierarchy into a relational table (Table 13). This would be necessary when using an RDBS as the persistent store for the Directory Information Base. In terms of the directory, the Distinguished Name (DN) of each entry is essentially the primary key for that entry.

..., ou=Applied Sciences, ou=Computing Science, cn=Eric Kolotyluk

Composed of a sequence of Relative Distinguished Names (RDN), the DN also represents the hierarchical location of the entry in the Directory Information tree (DIT). Conceptually one may simply view the Directory as one large relation where: each entry in the Directory is represented by a tuple in the relation, the DN of the entry is the (composite) primary key of the relation, and each RDN in the DN is represented by one of the candidate keys of the relation that forms the primary key.

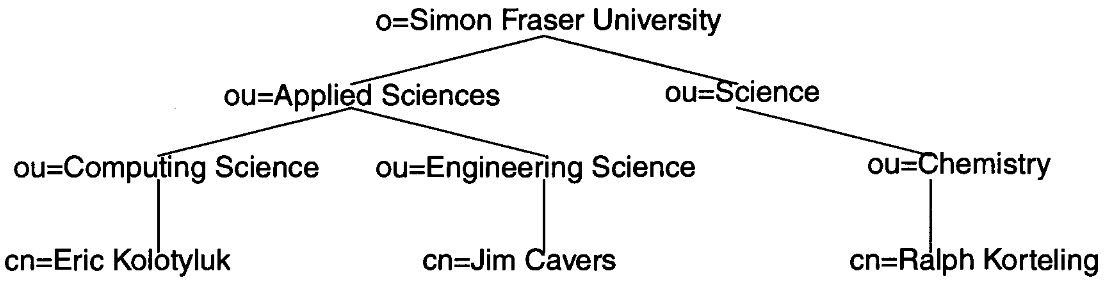
Next we consider the case where we want to map a given relational table in a Directory hierarchy (Table 13). This would be necessary for X.500 interoperability when presenting existing RDBS information via the Directory. In terms of the RDBS the attributes forming a candidate key of each tuple are used to determine a specific location in the Directory hierarchy.

(faculty, school), firstName, lastName

Given a database relation, we may want to map the tuples into entries of the DIT and vice versa. In the simple case this might look like:

Table 13: Straightforward Mapping

faculty	school	firstName	lastName
Applied Sciences	Computing Science	Eric	Kolotyluk
Applied Sciences	Engineering Science	Jim	Cavers
Science	Chemistry	Ralph	Korteling



In this case we might assume that we simply map the first attribute of all tuples in to the second level of the hierarchy;

faculty=Applied Sciences $\leftrightarrow$ ou=Applied Sciences

faculty=Science $\leftrightarrow$ ou=Science

and map the second attribute of all tuples into the third level of the hierarchy;

school=Computing Science $\leftrightarrow$ ou=Computing Science

school=Engineering Science $\leftrightarrow$ ou=Engineering Science

school=Chemistry $\leftrightarrow$ ou=Chemistry

To generalize, we map each attribute a_n of a relation into the corresponding level l_m of the directory and visa versa. In particular, in an automated system we might want to invent a notation to specify this mapping, for example,

ou=faculty, ou=school

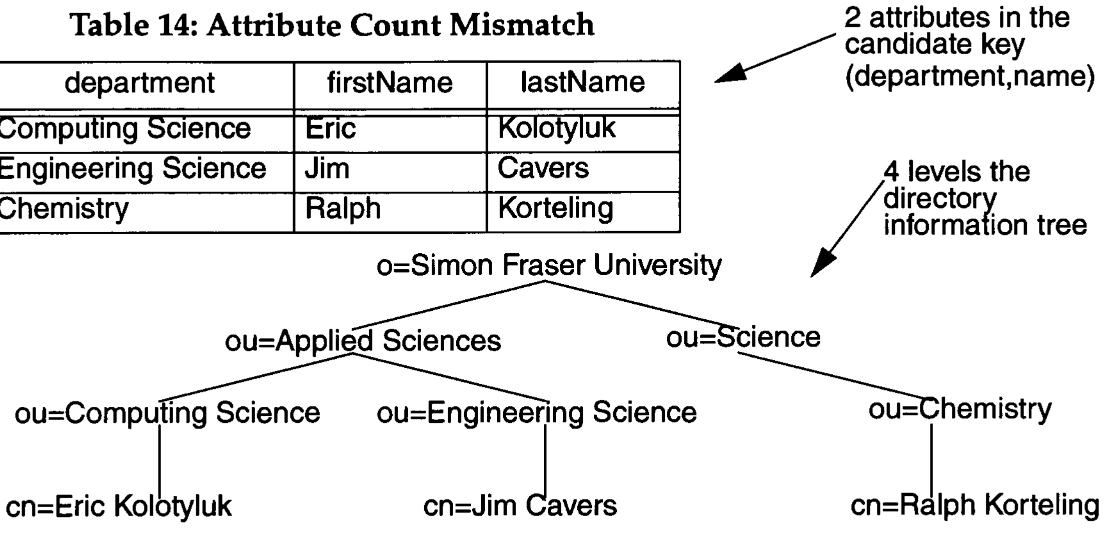
which means implicitly map the tuples with relational attribute values for faculty into directory entries with attribute values for organizationalUnit and the tuples with relational attribute values for school into directory entries with attribute values for organizationalUnit where entries with 'school' attributes are subordinate to entries with 'faculty' attributes.

This seems like an attractive mapping technique, but in practice, information structures are not always so well behaved. In particular the previous mapping notation fails us in the following examples:

1. There may be a mismatch in the number of attributes forming a candidate key in a given database relation and the number of levels in a DIT subtree.

Table 14: Attribute Count Mismatch

department	firstName	lastName
Computing Science	Eric	Kolotyluk
Engineering Science	Jim	Cavers
Chemistry	Ralph	Korteling

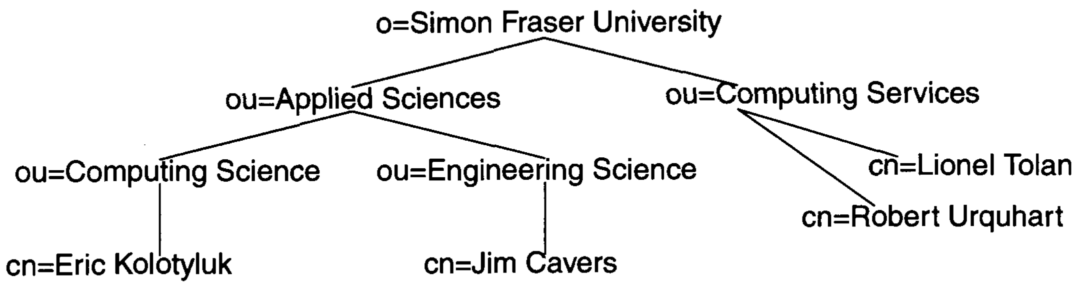


This can be a typical problem because often the designer of a relational database might choose one schema, say by department, and a directory administrator might choose a more flexible schema, say by faculty by department.

2. In the DIT, the depth of similar entries in the subtree can vary.

Table 15: Attribute Depth Mismatch

faculty	school	firstName	lastName
Applied Sciences	Computing Science	Eric	Kolotyluk
Applied Sciences	Engineering Science	Jim	Cavers
Computing Services	<<null>>	Lionel	Tolan
Computing Services	<<null>>	Robert	Urquhart

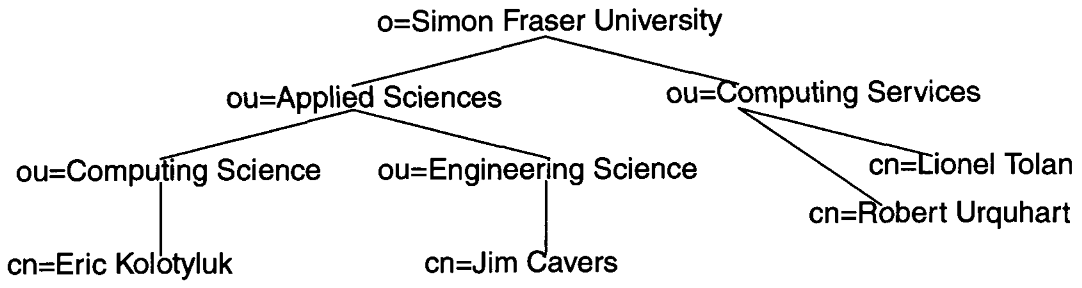


In real world hierarchies, organizational trees are rarely balanced. Given a composite candidate key with enough attributes to correspond to all levels in tree hierarchy, we can use nulls for attributes with no corresponding subtree entry. However, since the candidate key itself should never be null at least one attribute of the candidate key should be non null.

3. We can have a combination of 1 & 2.

Table 16: Attribute Count and Depth Mismatch

department	firstName	lastName
Computing Science	Eric	Kolotyluk
Engineering Science	Jim	Cavers
Computing Services	Lionel	Tolan
Computing Services	Robert	Urquhart



To handle these situations we need to formulate an effective strategy that resolves the problem and does not pervert the duality between the tree hierarchy and relations.

3.2.1 DN to Candidate-Key Maps

The basic approach to handling both count mismatches and depth mismatches is to maintain extra information in the form of an explicit map between DIT entries and the candidate keys in our relation. For example, we might extend the previous notation

```
ou=department
```

with the explicit information

```
ou=(Applied Sciences), ou=department(Computing Science)
ou=(Applied Sciences), ou=department(Engineering Science)
ou=department
```

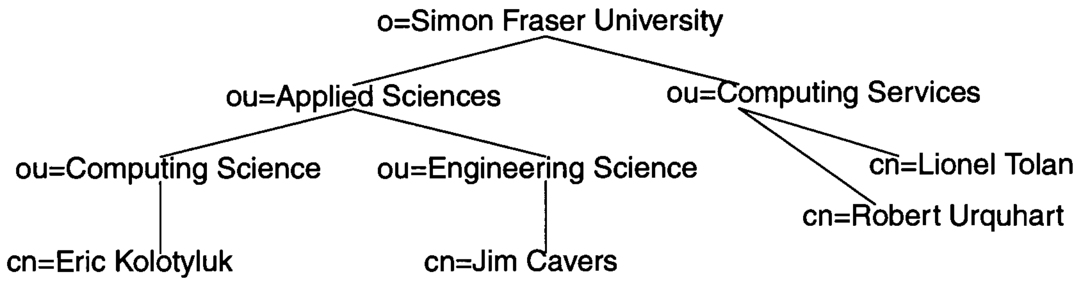
which means, group the relational tuples with attributes department=Computing Science and department=Engineering Science under the directory subtree ou=Applied Sciences. For all other tuples, simply map them to the Directory entries at the first level.

3.2.2 Directory Searching and RDBS Queries

When requesting searches in X.500, the scope of the search can either include the entire subtree or, by default, be limited to the children of the current entry in the directory. When X.500 is used as a front end to a relational database there needs to be some mechanism for translating a directory subtree search into an equivalent SQL query. Fortunately, this can be quite straightforward once a hierarchy map exists. Consider the following directory:

Table 17: Example Relation "FacultyPerson"

faculty	school	firstName	lastName
Applied Sciences	Computing Science	Eric	Kolotyluk
Applied Sciences	Engineering Science	Jim	Cavers
Computing Services	<<null>>	Lionel	Tolan
Computing Services	<<null>>	Robert	Urquhart



In this case there are three levels at which someone is likely to want to search:

1. The Simon Fraser University level, where the X.500 user would issue a query like

```
moveto "o=Simon Fraser University"
search -subtree sn=Kolotyluk
```

which would have to be translated to

```
select firstName, lastName from FacultyPerson
where lastName="Kolotyluk"
```

2. The faculty level where the user X.500 user would issue a query like

```
moveto "o=Simon Fraser University@ou=Faculty of Applied Science"
search -subtree sn=Kolotyluk
```

which would have to be translated to

```
select firstName, lastName from FacultyPerson
where lastName="Kolotyluk"
and faculty="Applied Sciences"
```

3. The school level where the user X.500 user would issue a query like

```
moveto "o=Simon Fraser University@ou=Faculty of Applied Sciences@
        ou=School of Computing Science"
search -subtree sn=Kolotyluk
```

which would have to be translated to

```
select firstName, lastName from FacultyPerson
where lastName="Kolotyluk"
and faculty="Applied Sciences"
and school="Computing Science"
```

In the third case it appears we could have left out the clause "and faculty=..." While this would have worked fine for the given data set, it would not have worked had there been another School of Computing Science in, say, the Faculty of Arts.

In order to automate this process two steps are necessary: (1) we must create a map from the Directory's Distinguished Name (DN) space to the various set of relation attributes needed to perform a search for a given DN, (2) given a Directory search request from a given DN, transform the Directory search into a relational (i.e., SQL) search. The first step of this process need not be very efficient as it happens only at registration time when the given Directory Service Agent (DSA) first starts up. The second step of this process is invoked any time an appropriate Directory search request is made and so should be as efficient as practical. The following two algorithms demonstrate these two steps.

The least general rule for mapping is simply an ordering of the relevant keys in the relation, for example (k_2, k_1, k_3). In this case we simply assume that the Directory attributes will have the same names and values as those in the relational database. To handle problems such as count mismatch or depth mismatch we also have other rules which map between Directory and relational attributes names and values.

Algorithm 1. Build a Hierarchy Map

Input: i) a database relation P, with *n* attributes $\{a_1, ..., a_n\}$

ii) a mapping describing how to map the relation into a tree with the levels of the tree ordered by attributes $\{k_1, ..., k_m\}$, where $k_1, ..., k_m \in \{a_1, ..., a_n\}$ and $k_i \neq k_j$ if $i \neq j$.

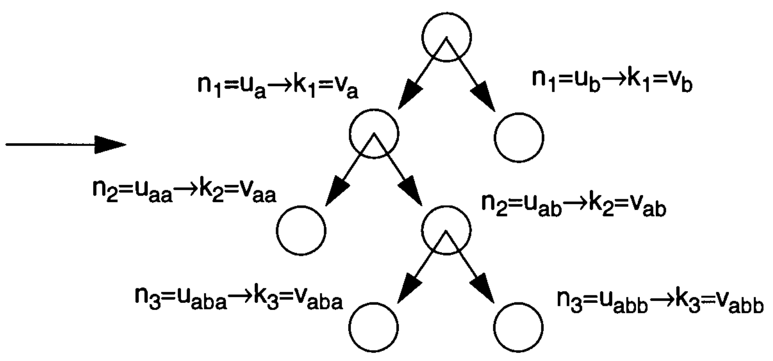
Output: A tree T containing mappings from the Directory's Distinguished Name space and Directory Information Tree structure to sets of relational database attributes.

Method:

```
DEFINE PartitionRelation(P, Level, MaxLevel)
    IF Level <= MaxLevel THEN
        sort relation P according to $k_{Level}$;
        partition relation P according to the values of $K_{Level}$ into $P_1$, ..., $P_{MaxLevel}$
            where m is the number of values of $K_{Level}$;
        FOR i := 1 to m DO
            P.children := $P_i$;
            $P_i$.parent := P;
            call PartitionRelation($P_i$, Level +1);
        END FOR
    END IF
END
BEGIN
    call PartitionRelation(P, 1, m); { m is the number of keys of $k_1$, ..., $k_m$};
END
```

*Explanation:*Mapping rules define a mapping between the hierarchical Directory name space $\{n_1, ..., n_m\}$ and relation attribute space $\{k_1, ..., k_m\}$. For simplicity we will ignore the Directory names and concentrate on the structure of the directory name space.

k_1	k_2	k_3	a_i	a_n
v_a	v_{aa}	v_{aaa}	v_i...	v_n...
v_a	v_{aa}	v_{aab}	v_i...	v_n...
v_a	v_{ab}	v_{aba}	v_i...	v_n...
v_a	v_{ab}	v_{abb}	v_i...	v_n...
v_b	v_{ba}	v_{baa}	v_i...	v_n...

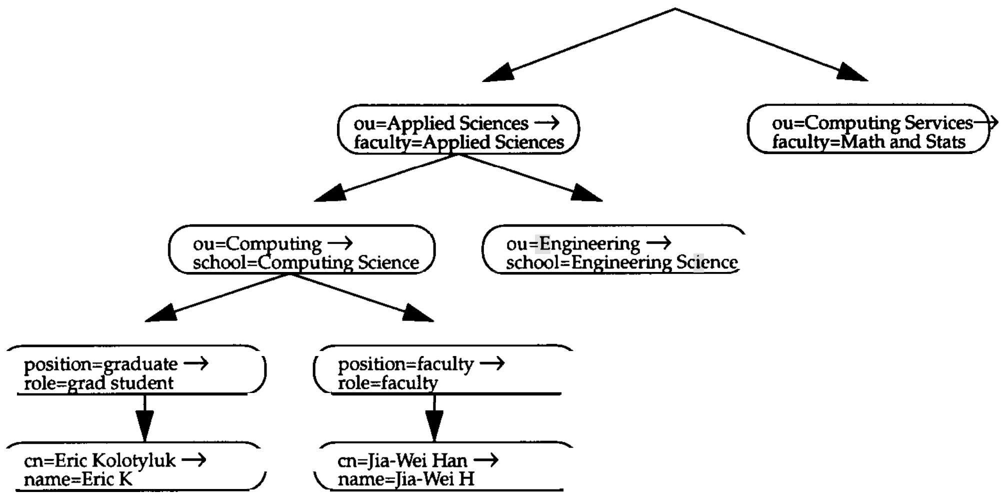

$n_1=u_a \rightarrow k_1=v_a$
$n_1=u_b \rightarrow k_1=v_b$
$n_2=u_{aa} \rightarrow k_2=v_{aa}$
$n_2=u_{ab} \rightarrow k_2=v_{ab}$
$n_3=u_{aba} \rightarrow k_3=v_{aba}$
$n_3=u_{abb} \rightarrow k_3=v_{abb}$

The essence of the algorithm is to successively partition the relation P into a hierarchy of 'key' attributes and their values, starting with the most significant attribute representing the highest level of the hierarchy.

Example: Given a relation P of people in a university environment,

faculty	school	name	area	role
Applied Sciences	Computing Science	Eric K	database	grad student
Applied Sciences	Computing Science	Jia-Wei H	database	faculty
Applied Sciences	Engineering Science	. . .	. . .	. . .
Computing Services	<null>	. . .	. . .	. . .

and a rule of the form {faculty,school,role} which declares the candidate key relevant to the mapping hierarchy as well as the order of the tree, we could form the following mapping tree.



Pass 1: partition the relation P using the key attribute faculty.

Pass 1.1: partition faculty=Applied Sciences using the attribute school.

Pass 1.1.1: partition school=Computing Science using the attribute role.

Pass 1.1.2: partition school=Engineering Science using the attribute role.

Pass 1.2: partition faculty=Math and Stats using the attribute school.

. . .

Algorithm 2. Build a Hierarchy Map Using Auxiliary Mappings

Input: i) a database relation P, with n attributes $\{a_1, ..., a_n\}$

ii) a set A $\{a_1, ..., a_x\}$ of auxiliary mapping rules to resolve count mismatches or depth mismatches. Each of these rules specifies the path of the tree branch affected as well as the explicit relational attributes to use.

Output: A tree T containing mappings from the Directory's Distinguished Name space and Directory Information Tree structure to sets of relational database attributes.

Method: Construct a mapping tree T based on the input syntax of the mapping rules.

```
    mappingRule ::= <<RDN>> <<relationalAssociation>> <<direction>
    relationalAssociation ::= empty or "(" <<RHS>> "=" <<LHS>> ")"
    direction ::= ":" or "," or ";"
```

Each node of the mapping tree corresponds to some entry in the Directory Information Tree (DIT) and must have a corresponding Relative Distinguished Name (RDN). Optionally the rule will include an associated RDBS attribute and value (RHS=LHS). Making this optional is how we deal with count and depth mismatches. The direction notation states whether the next rule is a child and that we should descend the tree (:), whether the next rule is a sibling (,), or whether we should ascend the tree one level (;).

```
BEGIN
    IF NOT endOfFile(Input)
    THEN
        T := currentNode := new(Node);
        LOOP
            currentNode.RDN := parseRDN(Input);
            currentNode.relationalAssociation := parseRelationalAssociation(Input);
            direction := parseDirection(Input);
            IF endOfFile(Input)
            THEN
                BREAK;
            ELSE
                newNode := new(Node);
                CASE direction OF

                ":"  :   newNode.parent := currentNode;
                         currentNode.child := newNode;
                         currentNode := newNode;

                ","  :   newNode.parent := currentNode.parent;
                         currentNode.sibling := newNode;
                         currentNode := newNode;

                ";"  :   currentNode := currentNode.parent;
                         currentNode.parent := currentNode.parent;
                         currentNode.sibling := newNode;
                         currentNode := newNode;

                END CASE
            END IF
        END LOOP
    output(T);
    END IF
END
```

Explanation: In Algorithm 1 we constructed a mapping tree based on minimal specification from the user or directory administrator. However, this mapping is not able to handle conditions such as 'attribute count mismatch' or 'depth mismatch.' In such cases it is necessary for the user or directory administrator to specify auxiliary mapping rules that explicitly define the mapping between directory scheme (tree structure) and relational schema.

Example: Given a relation P of people in a university environment,

Table 18: Attribute Count and Depth Mismatch

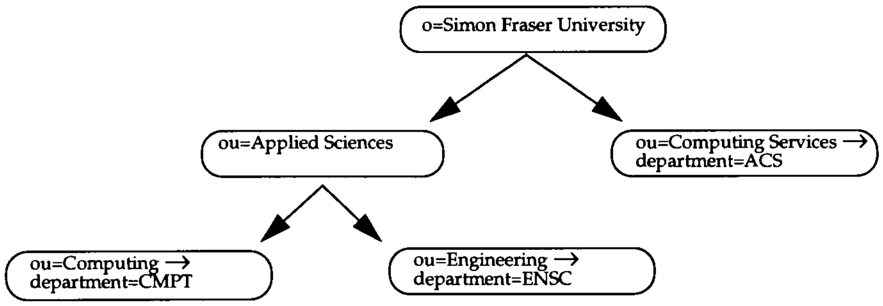
department	firstName	lastName
CMPT	Eric	Kolotyluk
ENSC	Jim	Cavers
ACS	Lionel	Tolan
ACS	Robert	Urquhart

and the auxiliary rules:

```
o=Simon Fraser University:
   ou=Applied Sciences:
      ou=Computing Science(department=CMPT),
      ou=Engineering Science(department=ENSC);
   ou=Computing Services(department=ACS);
```

we could form the following mapping

tree.



In this example the mapping of commonName (cn) attributes has purposely been left out in order to focus on the actual tree structure mapping.

Algorithm 3. Translate a Directory Search to an SQL Search

Input: i) the current directory position, a Directory Distinguished Name DN $\{N_1, ..., N_p\}$ where N_l is n_l=u_l and l is from 1 to p.

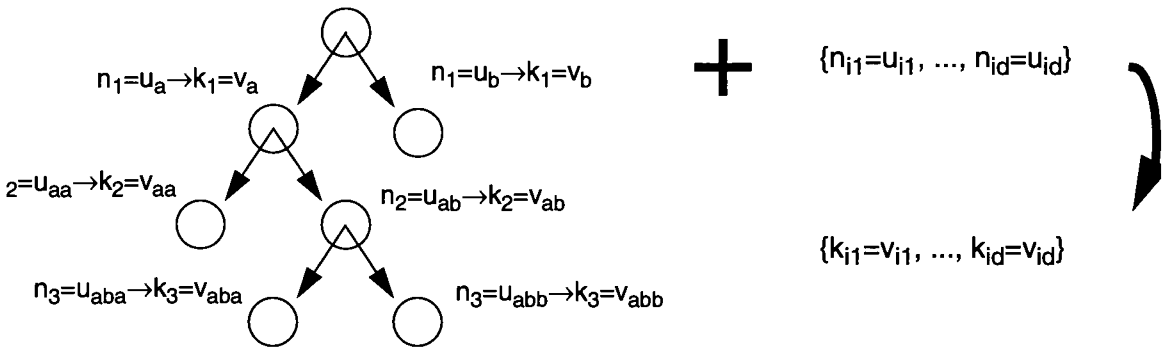
ii) a mapping from a set of Distinguished Names $\{DN_1, ..., DN_r\}$ to a set of candidate keys corresponds to $\{k_i=v_i, ..., k_j=v_j\}$ in relation P. This mapping is in fact the tree T built by algorithm 1 (i.e., hierarchy-to-relation mapping).

iii) a search expression E, used to further qualify the search.

Output: An SQL statement.

Method: Find a hierarchy map position based on the current directory position. This is performed by starting from the root and locating the position based on a Directory Distiguished Name DN $\{N_1, ..., N_p\}$ where N_l is n_l=u_l and l is from 1 to p. Suppose that the query expression generated at this point is E. Using the map information at that point, establish a set of conjunctive predicates in the SQL statement to restrict the search. The conjunctive predicates are built by traversing along the path of the tree..

```
BEGIN
    R := null; { Initially, set the conjunctive query expression to null. }
    Current := the current dirctory position;
    FOR i := 1 to r;
        IF there exists an attribute name for Current which is $D_i$
        THEN  add $k_i$=$v_i$ to R;
              Current := Current.name;
        ELSE  print "Error! There exists no such name"
    END FOR
    OUTPUT "select * from" P "where" R "and" E;
END
```



$n_1=u_a \rightarrow k_1=v_a$
$n_1=u_b \rightarrow k_1=v_b$
$2=u_{aa} \rightarrow k_2=v_{aa}$
$n_2=u_{ab} \rightarrow k_2=v_{ab}$
$n_3=u_{aba} \rightarrow k_3=v_{aba}$
$n_3=u_{abb} \rightarrow k_3=v_{abb}$
+
$\{n_{i1}=u_{i1}, ..., n_{id}=u_{id}\}$
$\{k_{i1}=v_{i1}, ..., k_{id}=v_{id}\}$

Explanation: This is simply traversal of the mapping tree T looking for the name N_i at each level i in the tree. The time to search the tree is proportional to the length of the distinguished name and the breadth of the tree. Each time a name is found the corresponding relational key attribute-value pair is used to form a conjunctive predicate of attribute-value pairs.

Example: Using the map from the example in Algorithm 1, and a current Directory position of {ou=Applied Sciences, ou=Computing}, we are asked to search for entries with area=database.

We look up ou=Applied Sciences in our map and add the attribute faculty=Applied Sciences to our conjunctive predicate. Next we lookup ou=Computing Science and add the attribute school=Computing Science to our conjunctive predicate. Finally we form the SQL statement:
select * from P where faculty="Applied Sciences" and school="Computing Science" and area="database"

3.3 Differences in Schema Content

The basic problem with schema content differences is that it is likely a given organizational database will not contain information that is generally found in the Directory schema. More importantly, such a database will not have information that may be required by the Directory schema. There are basically two ways we can handle this situation:

1. expand the schema of the relational database to match that of the directory, or

2. maintain the extra information in the Directory's own DIB.

3.3.1 Expanding the Relational Schema

In this approach, the X.500/Relational interface would automatically create an extra schema in the relational database to support essential information required by the Directory. For example, if the directory object class "sfuPerson" required that a person entry be able to have a telephoneNumber attribute, but the given relational database of persons does not have telephoneNumbers in its schema, a PhoneNumbers relation would be created as is appropriate.

The main advantage of this technique is that the RDBS interface (generally SQL) could have direct access to the new information. This would promote interoperability in both directions; from the Directory to the RDBS, and from the RDBS to the Directory. Also, since the format of the new information would be created by the Directory it would be maintained in a format that is most convenient for the Directory to use as contrasted to the situation where the Directory must be configured to utilize information formats (schema) designed by a database administrator without consideration of the Directory.

The disadvantages of this technique are generally those disadvantages listed in Section 1.3.1. In particular the impedance mismatch can be a problem. For example, in the case of Sybase where char and varchar datatypes are limited to 255 characters, the directory would be forced to use the datatype text for attributes such as AccessControlList (ACL) which are likely to be longer than 255 characters. However, ACL attributes are just as likely to be less than 255 characters so the extra overhead associated with using text datatypes must be implemented regardless. Finally, the format of the ACL attribute is specialized to the uses of the directory and would not generally be useful for access via the RDBS interface.

3.3.2 Augmenting Information via the DIB

In this approach the X.500/Relational interface is assumed to have access to its own DIB persistent store mechanism which would be used to augment the relational database. For example, if the directory object class "sfuPerson" required that a person entry be able to have a telephoneNumber attribute, but the given relational database of persons does not have telephoneNumbers in its schema, a corresponding entry in the directory would be created with the telephone number attribute, while the rest of the information would continue to reside in the RDBMS.

The main advantage to this technique is that, assuming the Directory already has its own persistent store, the information is already in a format that is most efficient for the Directory to access—in short there is no impedance mismatch problem. Furthermore, for specialized attributes such as ACL, efficient operation of the Directory will depend on the most efficient access to the ACL information.

The main disadvantage to this approach is that it does not promote interoperability from the RDBS to the Directory. Another disadvantage is that we assume that the Directory already has its own persistent store. However, even if the Directory has its own persistent store, there are still reasons not to use it. In particular, there are entity identification and synchronization problems which must be solved. First of all, for each tuple in the RDB, there must be a unique correspondence of the tuple primary key and the primary key (Distinguished Name) of the Directory entry. More importantly, there can be serious synchronization problems since each time a modification is made to the directory, it must be reflected in the RDB. However, each time a modification is made to the RDB, the Directory will be unaware of this change, which can lead to data and schema inconsistencies.

3.4 Notation for Mapping Relations to X.500

Developing a formal notation for defining mappings is not easy. On the one hand there is a strong desire to be concise, on the other, the notation should be readable and familiar to relational database administrators, directory administrators, or both. The ultimate goal of such a notation is that it could be used in registering a given relational database with a given Directory Service Agent.

3.4.1 Mapping Tuples to Entries

At the most basic level we need to map the tuples of a relation into directory entries. In addition to mapping tuple attributes to entry attributes, every directory entry must belong to some object class, so we map the name of the relation to a given object class.

$$objectClass = relationName\ (\ da_1{=}ra_1,\ da_2{=}ra_2,\ \ldots,\ da_n{=}ra_n\ )$$

For example, given a RDBS relation Parts(partNumber, partName, price, description), if we want to map each tuple into a directory entry belonging to the object class part we might use

```
part=Parts(partNumber=partNumber, commonName=partName,
                caPrice=price, description=description)
```

to denote this mapping.

While this seems straightforward, often the information in the table is not in the same format as the directory. For example, by convention in the directory the commonName attribute of a person includes the person's first and last name (i.e., cn=Eric Kolotyluk), but information in an existing database might have separate attributes for the first and the last name. We might use

```
person=People(cn=firstName " " lastName, sn=lastName)
```

to indicate that the directory attribute commonName is formed by the relational attributes firstName and lastName, and separated by a space.

Alternately, an existing database may have just a name field of the form "lastname, firstname" but we need to construct a surname attribute for our entries. To handle such a situation we need to have an extended notation which includes pattern matching and information manipulation operations. We might use,

```
person=People(cn=name, sn=rexp(name, "(*),*") )
```

to indicate that the directory attribute surname is formed by using a regular expression to remove all the text before the comma in the relational attribute name.

3.4.2 Mapping Primary Keys to Distinguished Names

Another important requirement of directory entries is that some of the attributes be distinguished so that we can form a distinguished name. In the simple case we might just extend the previous notation to indicate which attributes are distinguished. For example

```
person=People(! cn=firstName " " lastName, sn=lastName)
```

the exclamation mark is used to indicate that the commonName attribute is distinguished and thus is part of the distinguished name.

However, as was discussed in Section 3.2 we will likely want to be able to map the relational attributes into the structure of the Directory Information Tree (DIT). While a notation was developed there to express hierarchy mapping we can create more concise notation by using the already developed notation for mapping tuples to entries. To do this we treat distinguished names more explicitly. For example, if we want to map the relation People(firstName, lastName, department, phoneNumber) into directory entries, we might use the entry mapping notation

person=People(cn=firstName " " lastName, sn=lastName, ou=department)

and the Distinguished Name mapping notation

ou@cn

to indicate that the relation People is mapped into a two-level subtree with the organizationalUnit and commonName attributes forming that part of the distinguished name.

To handle less compatible information structures we might want to use a more explicit form of DN mapping. For example, consider the previous relation, but an organization with more than one level in the directory hierarchy such as SFU.

ou=Faculty of Applied Sciences@ou=Computing Science(department=CMPT)

ou=Faculty of Applied Sciences@ou=Engineering Science(department=ENSC)

ou=Academic Computing Services(department=ACS)

This gives the administrator more flexibility in mapping. We may also want to handle a relation like People(firstName, lastName, department, faculty) with an explicit form like

ou=Faculty of Applied Sciences(faculty=AS)@ou=Computing Science(department=CMPT)

3.4.3 Handling Multivalued Attributes

There are basically two ways we can express mappings for multivalued attributes by way of foreign keys. Given two relations, People(firstName, lastName, idNumber) and

Phones(userId, number), where userId is a foreign key on People.idNumber, we might use:

1. allow for embedded SQL in our notation, for example
   person=People(cn=..., phoneNumber={select number from Phone, People where user=idNumber}.
2. some specialized notation, for example,
   person=People(cn=..., phoneNumber={Phone.number: user=idNumber})

Using SQL has the advantage that anyone familiar with relational databases will also be familiar with SQL. Also, in a DSA implementation the specified SQL can easily be sent to the SQL server. However, if the SQL is specified incorrectly this can lead to problems which must be handled correctly.

Using a specialized notation has the advantage that it can be made more compact (because it is specialized) and that it can be expanded (correctly) into SQL for data retrieval. However, it is generally not good practice to introduce new terminology or notation when not strictly required, so we will use the embedded SQL instead.

3.4.4 Registration of Mapping Information

The main point of developing a notation for specifying schema mapping from relational database systems to the Directory is to be able to configure the Directory for access to the information in the RDBS. That is, to register the RDBS with the Directory. In Section 4.3.1 a more tangible form of this notation will be used to specify these mappings so that the directory may interoperate appropriately with the RDBS.

Chapter 4 Implementation Experience

One part of research is ideas, theories, conjectures, and so on. Another part is testing ideas, theories, and conjectures, looking at details, getting one's hands dirty, and resolving those ideas, theories, and conjectures in terms of the actual experience gained.

As Chapter 1 discussed there are a variety of reasons for using the Directory as a multidatabase front-end to relational databases, and as Chapter 2 discussed there are a variety of techniques of designing heterogeneous database systems in general. This chapter discusses an approach and implementation that is interesting in a few aspects:

1. Rather than viewing the Directory as specialized homogeneous database, it is viewed as a more generalized heterogeneous database.
2. Rather than trying to delegate an entire subtree of the directory [gda] to a new information source, the new information source is integrated with the existing Directory Information Base (DIB).
3. Reflection[1] is used to extend the semantics of a directory entry by means of an entity called a proxy to 'register' foreign databases with the Directory.

The overwhelming motivation for these features came from the author's experience both as a systems designer and as a systems administrator. All too often a greater burden is placed on the systems administrator to support a given design, than on designing a system which addresses the reality of systems administration. Such practice has led to not only increasing organizational dependence on a few highly skilled professionals, but also a bottleneck in acquiring and utilizing new technologies—most organizations cannot expect to hire more systems administrators whenever a new computer system is acquired.

[1] Reflection is the process of reasoning about and acting on the system itself [40,41,42]. To expose, or *reify* its internals, a reflective system embodies *reifiable* data that represents or implements the structural and computational aspects of itself within itself at the meta-level. Such data must be dynamically self-accessible and self-modifiable by the user program. Furthermore, modification by the user must be 'reflected' to the actual computational state of the user program—this property is termed as *causal-connection.* [quoted from 43]

4.1 Introduction to Proxies

In terms of the Directory, a proxy is a sort of meta entry that, from the user's perspective, is not really part of the directory information. In terms of the X.500-1993 specification, a proxy would likely be implemented as a *service entry.* Service entries are not generally accessible through the primary directory service operations (i.e, read, list, search) but have their own set of service operations. Since implementations of X.500-1993 are not generally available at this time, for now proxies have been implemented as regular directory entries.

The main distinguishing feature of a proxy is that creating one and modifying its attributes has the side effect of modifying the operation of the underlying directory system. A more traditional method for implementing such effects is through configuration files (e.g., Unix rc files, or in ISODE, tailor files) or back-door channels (e.g., Unix signals). The problem with these mechanisms is that they are either restricted to start-up time, in the case of configuration files, or that the directory administrator must login with specialized privileges (i.e., root) in the system running the directory. By communicating with the directory via manipulating entries and attributes, not only can operation be affected dynamically, but a familiar and consistent communication channel is used. This is likely why the 1993 version of X.500 introduces service entries and service attributes.

It is important to realize that a proxy is not just another name for a service entry. While it may be implemented that way, a proxy[1] is also an operational component of the directory. In particular, a proxy is an agent which represents a collection of information available from some foreign source which is to appear in the directory.

[1] from Webster's Ninth New Collegiate Dictionary, First Edition
proxy \'prä¨k-sē¯\ **prox•ies**
[ME *procucie,* contr. of *procuracie,* fr. AF, fr. ML *procuratia,* alter. of L *procuratio* procuration] (15c)
1: the agency, function, or office of a deputy who acts as a substitute for another
2a: authority or power to act for another
b: a document giving such authority; specif :a power of attorney authorizing a specified person to vote corporate stock
3: a person authorized to act for another: PROCURATOR— proxy adj

4.1.1 Proxy by Example

As an example of how proxies are used, consider an existing directory of cartoon characters:

```
organizationalUnit=Cartoons
                   ├──────────── commonName=Pink Panther
                   ├──────────── commonName=Daffy Duck
                   └──────────── commonName=Yogie Bear
```

Let's say a directory administrator had a request to list Disney cartoon characters under this directory, and that they had access to an existing relational database of Disney cartoons. Typically the administrator would write a program of some sort to extract the information from the RDB, transform it as appropriate to a form suitable for the directory, then load it into the directory. However, as was discussed in Chapter 1, maintaining the integrity of the two sources of information can be a problem.

Rather than maintaining two sources of information, the administrator might create a proxy entry for the database of Disney characters.

```
organizationalUnit=Cartoons
                   ├──────────── commonName=Pink Panther
                   ├──────────── commonName=Daffy Duck
                   ├──────────── commonName=Yogie Bear
                   └──────────── proxy=Disney,
                                      objectClass=proxySybase,
                                      server=CMPT,
                                      databaseUser=disney,
                                      databaseUserPassword=<hidden>
```

The extra information here (i.e., objectClass=proxySybase) is used to direct the Directory Service Agent (DSA) to where the external information is located and how to access it. Thereafter, the Disney characters would be listed with the other cartoon characters in this directory. If the information in the external database changed, this change would be automatically reflected in this directory. For example, from the user's perspective, this directory would appear as

```
organizationalUnit=Cartoons
            |------------ commonName=Pink Panther
            |------------ commonName=Daffy Duck
            |------------ commonName=Yogie Bear
            |------------ proxy=Disney
            |------------ commonName=Mickey Mouse
            |------------ commonName=Minnie Mouse
```

Whether or not the entry proxy=Disney would actually appear to the user is discussed later in this chapter.

4.2 Why Proxies?

In this project, the use of proxies was chosen for their conceptual simplicity. From a directory administrator's perspective the proxy represents a single point of registration for another information source, contrasted with delegated attributes as seen later.

From an implementation perspective proxies were easier to implement due to their self-contained nature, as opposed to making sweeping changes to the existing DSA code. Ultimately, an object-oriented approach to proxies was easy to develop, creating subclasses of proxies to meet the specialized needs of other information sources.

4.2.1 Delegated Subtrees

The first approach considered in trying to integrate information from relational databases with the Directory was to delegate an entire subtree to the relational database. This would have been more consistent with the architecture of the Quipu Generic Directory API (GDA) of ISODE [gda]. Basically the GDA mechanism allows the Quipu DSA to delegate an entire subtree of the Directory to another process. It is the responsibility of this other process to receive DSA requests and to perform them. No particular assumptions are made of this other process and the information base it represents other than that it be able to perform the operations requested. In fact, the first practical GDA application developed (called qb) was a disk-based DIB implemented using gdbm[1].

[1] GNU database management library, Free Software Foundation.

The GDA mechanism provides two important functions:

1. a defined interface to which developers can build an adaptor to a new type of information base without having to worry about too many of the details of directory operation.

2. a fire-wall so that if the process fails it won't take down the entire DSA.

While this mechanism was well suited to the disk-based DIB, the disk-based DIB was designed specifically to support not only an X.500 DIB, but a Quipu DIB. The primary difficulty in federating the Directory with existing information bases is that they have generally not been specifically designed to support an X.500 DIB.

4.2.2 Delegated Attributes

Another approach to introducing foreign information sources into the Directory was recommended via the IC-Tech@isode.com mailing list by John Farrell of the ISODE Consortium Inc. This is basically an extension of the existing QUIPU delegated subtree mechanism, but of finer resolution, down to the attribute level. In this approach the directory administrator would create an 'Application Process' entry to denote an external information source. To register the external information, the directory administrator would create skeleton entries in the directory and add attributes of the form delegated-Attribute=*binding-information,* where the binding information would detail the type of the delegated attribute and from where the attribute values were derived. For example:

```
organizationalUnit=Cartoons
       |
       |-------------- commonName=Pink Panther
       |-------------- commonName=Daffy Duck
       |-------------- commonName=Yogie Bear
       |-------------- commonName=Disney,
       |                    objectClass=applicationProcess & sybaseServer
       |                    server=CMPT,
       |                    databaseUser=disney,
       |                    databaseUserPassword=<hidden>
       |-------------- commonName=Micky Mouse
       |                    delegatedAttribute=Disney:artist
       |                    delegatedAttribute=Disney:conceptionDate
       |-------------- commonName=Minnie Mouse
                            delegatedAttribute=Disney:artist
                            delegatedAttribute=Disney:conceptionDate
```

The main strengths of this approach are greater consistency with the established QUIPU delegation mechanism, and greater flexibility in integrating external information sources. For example, the delegatedAttribute attributes of a given entry might point to different information sources. This approach also inherently addresses the problem of differing schema content in that the skeleton entries can maintain attributes not found in foreign information sources. Also, there is a better division of information in the directory.

The fundamental disadvantage to this approach is complexity, both to the directory administrator, and in the implementation. Consider:

1. Rather than creating just one proxy entry, the directory administrator must create an applicationProcess entry (similar to the proxy entry) and create skeleton entries. It is not clear how this would be done—by hand or by some other automated process?

2. The skeleton entries would take up storage space in the directory, in addition to the information in the foreign system. In particular, the delegatedAttribute attributes of each skeleton entry would likely be the same for each, resulting in redundant duplication. This problem could be addressed through the use of inherited attributes (a feature of QUIPU) so that the delegatedAttribute attributes don't actually exist, save for one parent entry with the inheritedAttributes attribute specifying inherited delegatedAttibute attributes. However, this only increase the complexity of the setup to the directory administrator.

3. It is not clear how synchronization of entries occurs. For example, if the foreign information source gains another cartoon character, how and when does another skeleton entry get created in the directory.

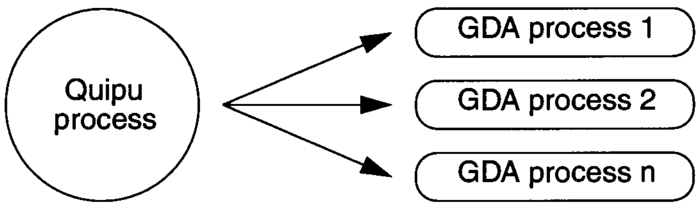
4. Implementing delegatedAttributes would likely require more intimate and wider sweeping changes to the DSA source code because of the wider number of features and mechanisms provided. Overall this would add to the complexity of the implementation.

The bottom line is that proxies were chosen to avoid complexity, but they don't offer all the features of delegated attributes. However, one reasonable approach would be to combine the two concepts by having a proxy mechanism which configures and manages the delegatedAttribute mechanism. Ultimately from the directory administrator's perspective this would offer a solution as simple as proxies, but more powerful than just proxies alone.

4.3 Implementing Proxies

The current implementation of X.500 directory proxies is in ISODE's Quipu GDA application as an enhancement to the existing qb disk-based Directory Information Base. ISODE was chosen primarily because it is the most widely recognized development environment for ISO OSI applications and systems. In addition, ISODE is extremely affordable to educational institutions.

The current architecture of GDA is that of main Quipu DSA process and one or more GDA processes. Each GDA process is responsible for a delegated subtree of the Directory Information Tree (DIT).



```
organizationalUnit=Entertainers
                  |-------------- organizationalUnit=People
                  |-------------- organizationalUnit=Animals
                  |-------------- organizationalUnit=Cartoons
                                                     |  objectClass=quipuDelegatedSubtree
                                                     |  delegate= GDA process 2
                                                     |-------------- commonName=Pink Panther
                                                     |-------------- commonName=Daffy Duck
                                                     |-------------- commonName=Yogie Bear
```

In this example the Quipu process is responsible for some portion of the world-wide X.500 directory tree including the branch organizationalUnit=Entertainers. However, one portion of this DIT, organizationalUnit=Cartoons, is delegated to GDA process 2. Note how Quipu uses information from the Directory itself to associate the subtree with the appropriate process.

The only GDA application distributed with ISODE[1] is the disk-based DIB. While there is opportunity to develop other GDA applications, it was decided to adapt the current qb GDA application to support proxies for the following reasons:

1. The proxy mechanism as previously discussed requires an existing DIB to resolve the schema content problem as previously discussed.
2. The GDA mechanism is ideal for development and testing because software failures are isolated to the GDA application and have minimal impact on the parent Quipu DSA process.

One serious problem with architecture however is that it ties the proxy mechanism to the qb implementation. As the proxy mechanism is really just an abstraction on the DIT, it would be more effective to implement it in the DSA process itself so that the proxy would operate independent of the underlying information base mechanism. However, implementing proxy support in the Quipu DSA process itself would add complexity to the implementation and would bypass the fire-wall that the GDA mechanism provides. Also, as the proxy mechanism is for the most part an experiment, it was more reasonable to minimize development effort until more experience was gained.

There are two significant components to the proxy mechanism: the proxy manager object, and the proxy objects themselves. At DSA initialization it is the responsibility of the proxy manager object to scan the Directory for proxy entries, identify the type of proxy, then create the appropriate proxy objects. Also, for all DSA operations, the proxy manager determines if a proxy is involved, identifies the appropriate proxy objects and passes on the request to them. Finally, if a result is returned, the proxy manager merges the results from the proxy objects with the results from the rest of the DIB.

[1] as of ISODE Consortium Release 1.2a, January 1994.

4.3.1 Registering Mappings

Using the mapping described in "Notation for Mapping Relations to X.500" on page 51, the proxy depends on AVAs to acquire the knowledge to perform the mapping from relational tables to directory entries. For this the following attributes are defined:

proxiedObjectClass, proxiedAttribute, and proxiedName.

Note that we decompose the notation slightly to structure the information as attribute-value-assertions.

For example, we might create a proxy:

```
proxy=CMPT-ENSC People
        objectClass= proxySybase
        server= CMPT
        databaseUser= quipu
        databaseUserPassword= <hidden>
        proxiedObjectClass= person=People
        proxiedAttribute= cn=firstName " " lastName
        proxiedAttribute= phoneNumber=
            (select number from People, Phones where user = idNumber)
        proxiedName= ou=Computing Science(department=CMPT)@cn
        proxiedName= ou=Engineering Science(department=ENSC)@cn
```

4.4 Implementation Challenges and Other Issues

Aside from the obvious challenges of resolving schema differences between X.500 and a RDBMS, there were a number of challenges in dealing with the X.500 model as well as the ISODE implementation of it.

4.4.1 Information Isolation

One of the biggest challenges of developing code in the QUIPU GDA environment is that there is no ready way to access entries in other parts of the Directory Information Tree (DIT). While this is technically possible, as a DSA is entitled to attempt to access any other DSA via the Directory Service Protocol, GDA just does not provide a function library to support external access. Another significant reason proxies were chosen was that it was straightforward to define the proxy in the same DIB as the proxied entries were to appear.

4.4.2 DSA Caching

In the original QUIPU implementation the entire DIB was read from disk into virtual memory. The GDBM-based implementation preserves this mechanism somewhat in that a cache of the most frequently used entries is maintained in VM. When the cache reaches a high-water mark, the least recently used entry is discarded.

While it would be tempting to use this mechanism for external information sources like the proxy mechanism there is a serious problem of supporting such replicated data. In the case of GDBM-based implementation, there is only one accessor to the database, the DSA. In the case of the proxy mechanism the DSA is only one of potentially many users accessing and modifying the underlying database. For this reason it was decided to sidestep the entire issue of replication and not support caching of proxied information.

4.5 Implementation Status

The current implementation of proxies in ISODE is still very minimal in many respects. It does not handle features such as access control lists and inherited attributes, it does not support compare and search operations, it is grafted onto the GDA QB back-end, and so wholly dependent upon it.

What the current implementation does provide is a demonstration of viability of using X.500 to front-end other information systems. While there are no formal performance results, casual use of the system seems no worse than that of the existing GDBM-based directory.

Chapter 5 Possible Future Directions

While it is hard to say what directions information management systems will be taking twenty, fifty, even one hundred years from now, it's probably safe to assume we will continue to see exponential growth in the amount of information stored year by year. Whether our frustration will increase, trying to find that piece of information we want, or whether we will ever truly become masters of information depends on how we shape the technology for storing and managing information.

In the preceding chapters we've seen how existing technology, X.500, can be shaped and extended to serve new purposes and to access information types beyond many people's expectations of the original design. Whether X.500 is the best design for a truly international information directory remains to be seen, but while we have a nascent and powerful model such as this we should see how far we can push the model before it breaks. When we push it past the breaking point, we may start to imagine what should come next to replace X.500.

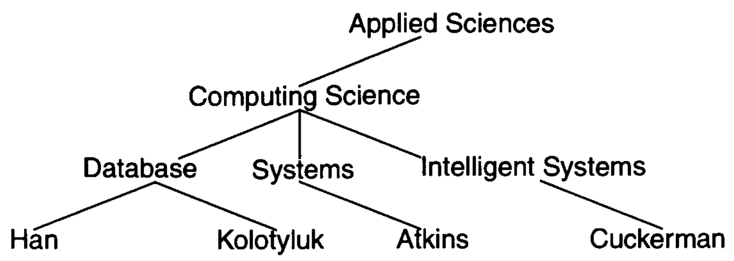
5.1 User Definable Hierarchies

In Chapter 3 we saw how a given relation can be mapped into a tree hierarchy, and that we can create a notation to define this mapping. However, what we did not explore was, for a given relation, how many mappings are possible. For example,

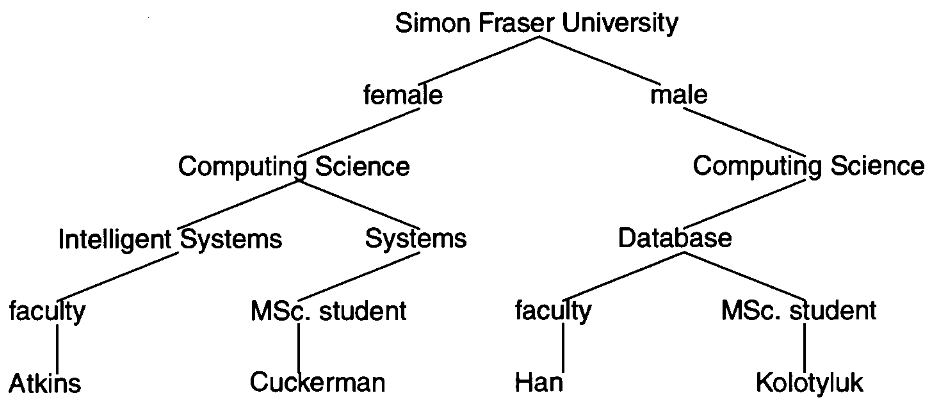
Table 19:

faculty	school	field	position	gender	name
App. Sc.	Comp. Sc.	Database	faculty	male	Han
App. Sc.	Comp. Sc.	Systems	faculty	female	Atkins
App. Sc.	Comp. Sc.	Database	MSc. student	male	Kolotyluk
App. Sc.	Comp. Sc.	Int. Systems	MSc. student	female	Cuckerman

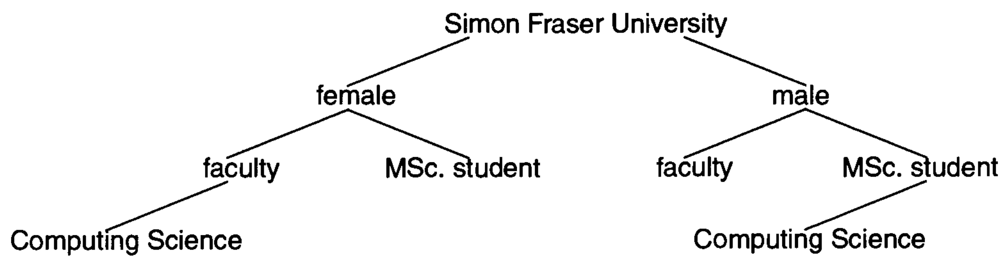
consider a database relation with a lot of demographic information. One way we might present this information hierarchically is:



However, while this might be typical of how a directory administrator might set-up a hierarchy, it is not the only possible hierarchy. Imagine a researcher in Women's Studies studying gender balances in higher education. She might prefer the following hierarchy:



or even



The point is, why should computer users be restricted to one view of the Directory? It is generally recognized that often people prefer to look for information by browsing hierarchical structures. Even though many times they have access to searching functions, like the Unix `find` command, they will choose to browse hierarchically. For example in Unix they would either use a sequence of `ls` and `cd` commands, or would use a graphical file system viewer to browse from directory to directory. This is not surprising when considering how often people are looking for something, but are unsure of what they are looking for, at least unsure enough to be unable to express it using search mechanisms.

Consider a more global example. Imagine that research papers were described by Directory entries. Such an entry would list, via attributes, the title, the authors, the abstract, the research areas, the location[1] of the full paper, and any other pertinent attributes. Typically these entries would be found at the originating sites, under that site's Directory hierarchy. For example, an entry named "c=ca@o=Simon Fraser University@ ou=Faculty of Applied Science@ou=Computing Science@ou=Library@paper=X.500 as a Heterogeneous Database" might have attributes: field=computing, area=database, keyword=heterogeneous. Someone looking for papers in this area might decide to search the directory for entries based on those attributes. The problem is:

1. Searching the global Directory could take an enormous amount of time, and chain the search operation to thousands of other Directory Service Agents all around the world.
2. Most DSAs do not allow searching from the Directory root because of (1). In general, most DSAs limit searches to the organization level and below.
3. As mentioned before, searching is not always appropriate when people have only a vague notion of what they are searching for.

What might be preferable is that the Directory support an additional hierarchy from the root, for example field=computing@area=database@ou=Masters Thesis@paper=X.500 as a Heterogeneous Database might name a directory entry which is an alias to the previously described entry. This hierarchy would have tremendous advantage to someone who wanted to browse a given field for papers. As attractive as this new scenario is, what is the correct hierarchy to define for research papers, and who's to say there is only one appropriate hierarchy? Why not let Directory users define new hierarchies on demand?

[1] The location of the actual paper itself might be described by attributes, for example, the attribute ftp=ftp.cs.sfu.ca/papers/database/tech-report-23.ps. However, as we have seen the directory can also map external data sources into attribute values, so we might just as well define a PostScript attribute. When the value of this attribute is read, the directory would return the full PostScript text of the paper—let the Directory handle the actual storage or acquisition of the paper text. The paper entry might also have an SGML attribute which might be a multimedia, hypertext version of the paper.

5.2 Indexing Servers

One of the most useful of new internet services is the Archie application. Basically, Archie is an indexing service which routinely scans the wealth of internet ftp sites for keywords to index. When someone is looking for a file in a particular subject area they can request the Archie service to provide a list of ftp sites based on keyword search. Many Archie user applications will provide a graphical browser of all such sites and allow the user to even browse the ftp hierarchy of each listed site and retrieve selected files.

A related service, Veronica[1], indexes Gopher space, the collective known services running the Gopher servers. Similarly, another service, Jughead[2], also indexes Gopher space, but just the high level directories. ALIWEB[3] is a first attempt at indexing World-Wide-Web (WWW) space, the collection of servers which Mosaic is based on.

The power of Archie and friends lie in the fact that they build indexes. It would be possible to implement a similar sort of service via the Directory by having DSAs which also maintain such indexes via Directory alias entries. Such DSAs could also structure these indexes hierarchically so that they could be browsed. This is something that is not possible with Archie. You search on a keyword and receive a flat list of ftp sites matching the search.

It is quite likely that a Directory-based Archie-style service would be much more powerful than Archie. In particular, information found in the Directory is much more structured through the use of attributes and object classes. This would likely increase the efficacy of index building. Also, multidimensional indices could be built and maintained as attributes. As we've seen, given a set of attributes, we can construct a number of different organizational hierarchies. Building and maintaining such multidimensional indices would make it possible for a DSA to be easily reconfigured to present a variety of hierarchies leading to more flexible browsing.

---

[1] Very Easy Rodent Oriented Netwide Index to Computerized Archives.

[2] Jonzy's Universal Gopher Hierarchy Fxcavation And Display.

[3] Archie-Like Indexing for the WEB.

Another advantage to this scenario is that as has been shown in this Thesis, the Directory can also be used to provide a gateway to other information sources, such as relational databases. By bringing such information sources into the fold of the Directory we can bring such information into the indexing mechanisms previously described. In fact, we might even use a RDBS to manage the indices for a given DSA.

5.3 More Powerful Indexing Servers

Consider a world-wide X.500 service tying together information systems for practically every organization in the world. As useful as this sounds, it is likely that the chosen physical organization of world-wide hierarchy would be constructed according to political and administrative goals rather than research or academic. However, a researcher looking for information will likely have a preconceived notion of what hierarchy would be most useful for his/her purposes. This section explores one scenario for implementing powerful indexers capable of easily building custom hierarchies.

For convenience, we will limit the scope of this scenario to academic research papers. First we assume that there exists an agreed upon standard for describing research papers via Directory attributes based on some ontology or taxonomy. This taxonomy itself might be maintained via the Directory to enable academics to more easily and thoroughly classify their work. This might result in some well defined organizational attributes such as objectClass=Research Paper, researchDiscipline=Applied Science, researchField=Computing Science, researchArea=Database, as well as a catch-all attributes such as keyword=heterogeneous, keyword=federated, and keyword=interoperability.

The basic process for indexing this information would be:

1. Walk the world Directory Information Tree. At every level of the DIT, search for research paper entries (i.e., objectClass=Research Paper).
2. Index each such entry in a relational database based on the prescribed taxonomy of attributes.

3. Construct one or more new Directory Information Trees based on a given hierarchy map and decorate the new tree(s) with alias entries pointing to the real entries found during the world tree walk.

5.3.1 Walking the World Tree

This is potentially the most expensive part of the entire process, although very easy to program. Ideally we would start at the root of X.500 directory and start a search operation searching for all occurrences of entries with objectClass=Research Paper. After all, this is what the Directory was designed for. Unfortunately, most DSAs enforce service limits. In particular, top DSAs, such as for a country, will not permit an entire subtree search. In this case we simply use a tree-walk algorithm until we get to a level of the tree where we can directly use a Directory subtree search. One other potential source of problems is that some DSAs might implement anti-dredging controls which could interfere with our search.

Walking the entire world DIT more than once a week should not be necessary, and could be done as seldom as once a month. Walking the tree more than once a week could potentially be disruptive to some site's DSAs as had been the case with some ftp and gopher sites which suffer from indexing services such as Archie and Veronica.

DIT walking might also be spread out over the day to minimize impact by restricting the walk to off hours. This could be easily achieved through a number of means, such as using country information to determine time zone.

5.3.2 Building the Master Index

For every target entry found (i.e., objectClass-Research Paper) a tuple in a relational database would be created. Once completed, this relational database would be the master index, and would look something like:

Table 20: Research Papers Master Index

DN	research Discipline	research Field	research Area	keywords
	Applied Sciences	Computing Science	Database	heterogeneous, federated, interoperability
	Liberal Arts	History	Military	strategy, tactics, weapons
	Science	Chemistry	Polymer	
	Liberal Arts	Psychology	Infant	language, reasoning

where the DN attribute of the master index would identify the original Distinguished Name of the entry that is indexed. The high-level attributes researchDiscipline, research-Field, and researchArea would follow the strict taxonomy previously agree upon. The remaining low level attributes such as keyword, would further aid in searching, but would offer more flexibility beyond the standard taxonomy.

The use of a relational database for the master index is important for a few reasons:

1. Current RDBS technology is fairly mature and highly effective at organizing and accessing information in this form.
2. As we have seen, it is straightforward to create a mapping between a relation such as this and a hierarchy such as the Directory Information Tree. It is even easier when the information structure is well behaved as in the case of a prescribed taxonomy as indicated here.
3. As we have seen, there are a variety of hierarchies which can be defined from a given relation. This is especially important when creating custom tailored hierarchies.

5.3.3 Constructing a New Custom Hierarchy

Once the master index is constructed we can proceed to define and build whatever new Directory hierarchy we wish. Definition might happen once, statically where some organization would maintain the standard research paper taxonomy hierarchy and so define a standard relational to hierarchy map. Thereafter, each time a new master index is constructed (say once per week), a new branch of the Directory Information Tree would be constructed as per the map. This would be the reverse process of the world tree walk, where each tuple in the master index would be placed in the new DIT as an alias for it's original entry. Alternately, we might never physically construct a new DIT, rather the map is used to transform all Directory queries into SQL queries against the master index. This would save the trouble and storage space of physically maintaining a separate subtree, with the small overhead of query translation on every directory operation against the implicit directory subtree.

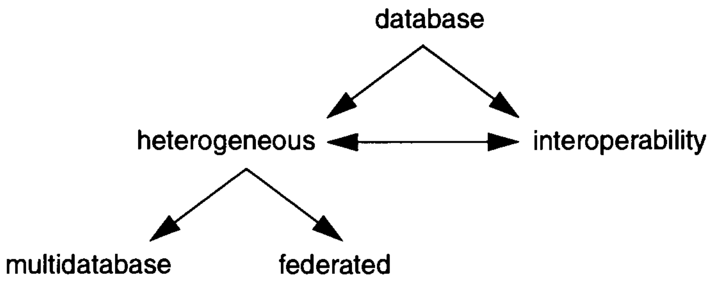
To appreciate the true power of the master index, consider the case where a researcher wants to define their own research hierarchy. Consider a user customizable DSA where the researcher registers their own hierarchy map against the master index relation. Once this map was registered, they would query the Directory via standard DUA tools, but see their own custom hierarchy. For each directory operation they initiated, the operation would be dynamically translated in master index query, returning the DN stored in the index being reflected back as directory aliases, which would ultimately be dereferenced back to the original research paper entries in the world DIT.

5.4 Concept Hierarchies

How many times have you heard someone say, or caught yourself thinking, something to the effect of "stupid computer, do what I want, not what I say!"? This kind of frustration is often encountered when dealing with information systems where a user makes a query to search for information on some topic, but only receives a portion what they really want, and often knows there is more information that they are looking for. But how can an information system find more than what you ask for.

Another emerging technology in computing science is 'knowledge discovery' (or 'data mining') where information systems do try to deliver more of what you want, in addition to what you ask for. An important aspect of knowledge discovery is the use of Concept Hierarchies [45] to provide a framework for searching for information. The concept hierarchy can be used as a sort of pattern matching template when performing searches for information with particular attributes.

As an example application of this technology, consider the claim made at the beginning of Chapter 1 that many of the reference papers used in this thesis were obtained over the network using Mosaic. While this was particularly convenient, it was necessary to make at least four separate searches on keywords like 'heterogeneous,' 'federated,' 'multidatabase,' and 'interoperate.' Each search returned a wealth of references not related to thesis, for example, 'heterogeneous network systems' and 'interoperability of network protocols.' In Chapter 2 a survey of the field revealed the taxonomy of the field of work in heterogeneous database research. Using this taxonomy we might construct the following Concept Hierarchy:



If you were to query a knowledge discovery system, with the given concept hierarchy, to search for articles on 'database, heterogeneous' or 'database, interoperability' it could return articles on multidatabases and federated database by induction on the concept hierarchy.

Knowledge discovery relates to X.500 in a couple of ways:

1. The Directory supports the notion of 'imprecise matching' when searching for entries. Typically this can be implemented by simple pattern matching such as regular expressions or by soundex algorithms. Matching by concept hierarchies would also be a useful way to search the Directory.

2. Where would the Directory find such concept hierarchies? Why not in, or through, the Directory itself? Because the Directory is a global 'open system,' it would be of enormous benefit if such concept hierarchies were available to all.

Furthermore, such concept hierarchies need not be static, for we might store the concept hierarchy in a more abstract form such as a relational database, then map a given concept hierarchy into a directory hierarchy for user browsing. The user might also instruct the Directory, via the proxy mechanism, to formulate new concept hierarchies which would in turn affect the directory's imprecise matching mechanism. Such mechanisms might be used in concert with the dynamic hierarchy matching described earlier, for in a sense, specifying a map from a relational database to the Directory hierarchy is like specifying a concept hierarchy.

5.5 The Universal Relation

In [11] Ullman introduces the notion of 'The Universal Relation.' The basic idea is to forgo the notion of normalization of a relational schema, at least at the user interface level. As there is substantial practical and theoretical reasons for normalizing relations, this can likely hidden to the user and maintained by the RDBS itself.

The basic advantage to the user of the Universal Relation is the simplification of queries. Rather than phrase a query in terms of relations and joins, the user frames the query in terms of attributes only. For example, using:

```
SELECT NAME
WHERE PHONE_NUMBER = '291-3014' ;
```

as opposed to:

```
SELECT NAME
FROM PERSON,PHONE
WHERE PERSON.personId = PHONE.personId
AND PHONE_NUMBER = '291-3014' ;
```

is not only much more simple to type, but conceptually easier to formulate.

In [14], one of the first papers to discuss such an interface, the approach significantly depends on the proper naming of attributes. Under such assumptions, the naming chosen in Table 9, "People," and Table 10, "Phone," on page 34 would not have worked well in the previous example. However, such naming is only a mapping issue and techniques such as Sybase's primary-key and foreign-key tables could be used to accommodate the Universal Relation model on existing database implementations. The point is, much can be done to utilize higher level data models on existing databases.

In a sense the X.500 data model is quite close to the Universal Relation model. In X.500, the search operation only specifies attributes. There is no notion of relations, projection and joins, or of normalized schema. Such notions can be relegated to lower levels of the information system.

In [14] Carlson and Kaplan showed that proper naming of attributes can (in their case, must) be used to support the Universal Relation model by providing join information. Another characteristic of X.500 is that its schema (including attribute names) is highly formalized, and often highly descriptive. In the context of using X.500 as a front-end to existing relational databases, this strictness in attribute naming could be of use in automating the process tying together user visible X.500 attribute names, underlying RDBS attribute names, and join information for normalized relations. The point here is that if we hope to create Archie-like indexing and browsing systems for relational database systems front-ended by X.500 DSAs, we need to keep the user interface simple, and the administrative demands minimal.

5.6 The Open Systems Director

The Open Systems Director (OSD), or just Director[1], is a concept for a computer application that the author invented several years ago. Basically it extends the concept of file system utilities like Unix `ls`, `cd`, `rm`, `rmdir`, and like Apple's Finder for the Macintosh, to that of the X.500 Directory. In addition to being a super browser, the Director assumes that everything reachable by computer systems and computer networks is also reachable via the X.500 directory. Such an assumption is based on the concepts explored in this thesis, so that not only relational database information would be reachable

[1] In a sense, the Director is just that, it directs you through the Directory.

through the Directory and the Director, but user files, user preferences, system resources, research papers, stock market quotes, corporate product lists, etc.

Another way of viewing the Director is similar to applications like Gopher and Mosaic where users can browse globally for all sorts of information. Additionally the Directory would provide users with searching capabilities similar to Archie and friends, but in one integrated application. By utilizing the kind of multidimensional indexing or knowledge discovery techniques described previously, the Director would also allow users and systems administrators the ability to easily create new information hierarchies, or 'information spaces,' more appropriate to their needs. Furthermore, by utilizing attributes and indices, the Director would make it easier to restrict the user's view of Directory space as an aid to browsing and searching. The user might even want to define multiple such views and store them via the Directory for easy reference later.

Another significant benefit to the user is that the Director user interface would be based on the Universal Relation model, a model that is far more simple conceptually for the user than the normalized relation model. Given that the X.500 model closely matches the Universal Relation model, it should not be difficult to design the Open Systems Director this way.

Another important aspect of the Director would be the graphical user interface similar in some ways to the graphical file system browser found on most contemporary computer system, but more able to handle the diversity of information systems as in the InterBaseView Graphical User Interface [28].

5.7 The Information Superhighway

One of the notions the popular media has characterized of late is that of the Information Superhighway[1], a vast network of information destinations traversable at near the speed of light. This, evidently, is to be the next major infrastructure our global civilization is to become economically and socially dependent upon.

[1] US Vice President Gore is often credited with actually coining the term

A significant distinction between the Information Superhighway and any previous transportation system we've yet created is the destinations we are likely to visit. There are two facets to this distinction:

1. In the current global transportation network most of us each year are likely to visit less than a few destinations globally, and maybe a dozen or so regionally. Via the Information Superhighway, we are likely to visit hundreds or even thousands of different destinations globally each year.
2. Due to time and economic constraints we likely choose our destinations on the current transportation network very carefully, and plan our travel itinerary with equal concern. On the Information Superhighway we are likely not going to care much where we go, or in what order we visit our destinations. Rather, we will only care what is at each destination, and will not likely care about the destination itself.

The point here is 'road maps.' Current maps of the transportation system show destinations from a geographic point of view. Typically the most important features of a map are 'how you get there from here,' how far apart the destinations are, the political boundaries, etc. On the Information Superhighway we will need not only new maps, but a new kind of map.

This new road map will not be static. It will have to be dynamic as information destinations and pathways are likely to change daily. This new map will have to be interactive in that it will point out destinations not in terms of where they are, but in terms of their feature (or attributes). We will not want to go to 'X', but we will want to go to a place, any place, or perhaps all places that have 'X.' In these terms, the X.500 Directory might make a nice map.

This new road map, however, will be fiercely more sophisticated than any map previously created. The trouble is, it will likely be beyond the mastery of all but a few highway experts or cyberpunks[1], so good guides and chauffeurs will become indispensable. The kind of guide or chauffeur we will need will probably be something like the Open Systems Director.

[1] The term 'cyberpunk' is taken here to mean someone who is not only technically gifted enough to travel the information superhighway, but actually takes intellectual pleasure from the experience. This is meant to mean someone who is willing to put an above average amount of effort into mastering the tools of navigation and inordinate patience into pursuing what they are looking for. It is the author's opinion that we should not all have to become cyberpunks to navigate the Information Superhighway.

Chapter 6 Conclusion

The main motivation for developing this thesis was finding better ways to find and access information. As an academic, the author has long been fascinated with directory systems, and as a user, equally frustrated by their limitations. As a seasoned computer systems administrator, there has been a growing concern with the increasing complexity and variety of information systems and information management techniques. As an experienced systems developer, there is a profound scepticism that we need to suffer this much complexity and variety, as well as a profound belief that integration of technology is the best means of simplifying information management. As a student, there is a sense of awe at the richness of the database field, in particular, the ambitious efforts of other heterogeneous database researchers—clearly what is an important subject.

The author sees X.500 as an important new technology that has not been fully exploited, and even less recognized. In particular, X.500 is popularly seen only as an electronic mail address directory. In a few more cases, it is seen as a more general directory, but still a limited or specialized form of database. What is often overlooked is that the X.500 standard does not specify a database architecture so much as an information model. It is this perspective that is so tempting to exploit: To see X.500 as not only an opportunity to create a world-wide directory, but an opportunity to create a world-wide information model which can be used as a standard for interoperating with a wide range of other information systems.

X.500 is likely not the best information model for world-wide use, and we will probably find a better model someday, but such a new model should be significantly better lest we create yet another model for the sake of creating yet another model. In the meantime, X.500 provides the opportunity to experiment with a whole new range of information management applications, on a world-wide scale. While applications like Gopher, Archie, and Mosaic are flourishing in their novelty, they are limited by the underlying information models they depend on. X.500 provides an opportunity to do far more, using an existing information model as well as an existing operational infrastructure.

6.1 Summary of Results

One of the first challenges was to take a subject as broad as X.500 interoperating with relational databases and start to categorize the various related issues. Sometime during this process it became clear that some things could be generalized and abstracted to a higher level. For example, while a distinction is made between using X.500 to view raw relations (Section 1.3.2) and integrating X.500 with existing databases (Section 1.3.3), the process of viewing raw relations is merely a starting point and ultimately evolves into the final integration of X.500 with an existing database. While there is however a larger distinction between using X.500 to access relational databases and using relational databases as a persistent store for X.500, even this distinction disappears when one considers the whole context of information interoperability.

6.1.1 Three Issues of X.500-RDBS Interoperability

In terms of X.500 interoperability three main issues were explored:

1. Mapping Multivalued Attributes. Because X.500 deals with multi-valued attributes explicitly and RDBSs support multivalued attributes implicitly, there are many variables to consider in a proper mapping. Fortunately, more modern RDBSs which utilize SQL-2 or SQL-3 will have more explicit support for multivalued attributes, making easier opportunities for X.500-RDBS interoperability.

2. Mapping Between Directory Hierarchy and Relation Tuples. For political, operational, and administrative reasons the Directory Information Tree is one of the most prominent aspects of X.500. However, the flexibility of relational systems makes information hierarchy less prominent in that one can easily view the information by whatever hierarchy is convenient (e.g., SQL's 'group by' clause). Consequently, there are compelling reasons to incorporate this sort of flexibility into X.500.

3. Differences in Schema Content. There are two distinct approaches to dealing with differences in schema content: extend the relational system or keep the differences in the Directory's specialized information store. In practice, a combination

of both techniques is most pragmatic. Ultimately, both techniques should always be supported and the decision of which to use should be left to the discretion of the Directory/RDBS administrator.

However, these three issues are by no means exhaustive of the problems to be solved in X.500 interoperating with relational databases or any other type of database. In particular, the issue of Entity Identification is important to the issue of schema content, but has not been considered in depth. Furthermore, issues such as query processing and optimization, and transaction processing have not really been considered. In spite of the limits on the scope of investigation, it was still possible to construct a prototype system for viewing the contents of raw relations through the X.500 Directory.

6.1.2 Interoperability Mapping Notation

The notation for mapping relational schema to X.500 schema was developed as a necessary formalism for registering an external relational database with the Directory. In particular it was designed to be simple to implement and to use, especially in the context of defining mappings via the Attribute-Value-Association (AVA) model of an X.500 information entry.

The basic approach taken was to map the external information source into a common information model (the Directory's). This information model is roughly object-oriented as well and having similarities to the Universal Relation Model (Section 5.5) which seems to be comparable with any other model invented, but without having to invent yet another canonical model. On the other hand, the notation used is very primitive as compared to research efforts like Interbase, Myriad, ORECOM, IDEF-1X, EXPRESS, NIAM, OSCAM, and SchemaLog.

6.1.3 Registering External Information Sources Using Proxies

During the actual implementation of a prototype X.500-RDBS federation, the many details of contemporary software management made for slow progress. Nonetheless, the process of designing and implementing a small working system revealed several important design issues related to registration of external information sources:

1. Proxies and Reflection. When reasoning about meta-level issues such as registration, reflection techniques and information proxies offer a framework for designing the registration system.

2. Proxies vs. Delegated Attributes. While the concept of Proxies was useful for a simplified implementation and user interface, it is not completely compatible with Quipu's intended model of delegated attributes. Both models offer advantages and disadvantages, but not exclusive to each other. A workable integration of both models is both likely and desirable.

3. Inband vs. Outband Registration. Should the registration of external information sources and definition of the necessary mapping requirements be done inband via the Directory's own information model and operations suite, or should it be done outband behind the scenes using specialized custom interfaces? This author believes profoundly that inband registration is critical to flexibility and portability of information system interoperability.

4. When designing an inband registration system, there are clear inadequacies in the current implementation of ISODE's Quipu Directory. In particular, at the lower levels of the DSA to DIB interface there needs to be easy access to the rest of the Directory Information Tree (DIT). While a DSA must naturally have access to the Directory Service Protocol (DSP), the Generic Database API (GDA) of Quipu does not support access to the DSP interface. Both the Proxy and the Delegated Attribute models demonstrate this need in order to access registration information external to the system being registered.

5. Object-Oriented techniques were useful both in the design of proxy directory entries, and in the actual implementation of proxy objects within the DSA.

While there were many other important issues discovered in the prototype implementation (e.g., RDBMSs should be extended to support nested attributes and universal relation models), issues of registration were most profound and still need much attention.

6.2 Problems with X.500

As much as this work has developed the notion of using X.500 as federated information systems manager on a world-wide basis, there are problems with this approach.

6.2.1 Schema Control

X.500 has a great deal of machinery to support and control well defined information schema. Schema controls affect which attributes an entry may or must have, the format of the attribute values, whether or not an entry may have child entries, and even what object classes of entries may appear in a given subtree. As well defined and controled the Directory's schema is, there is a large price which is paid with all this control.

1. There is an enourmous burden on the Directory administrator to ensure that all the schema definitions are configured properly.
2. There is often inconvenience to the Directory user who attempts to add information to the Directory but is thwarted schema controls which refuse the addition of nonconforming information, but offer little or no help in defining what conforming information should look like.
3. The is a great deal of overhead to the developer adding new features to the Directory and trying to understand and utilized the internal details of the schema controls.
4. The problem with integrating the Directory with relational databases is that there is no east to use mechanism in place to automatically and effectively define new schema. This can be a paricular problem when mapping relational attributes to directory attributes—often the appropriate directory attributes do not exist. Defining new attributes is an onerous task: allocating object identifiers, defining new attribute classes, defining new attribute syntaxes and writing syntax translators, etc.

To deal with these problems there needs to be much more work done on making schema management, to coin an overused term, *user friendly.*

6.2.2 Design Limitations

X.500 was not designed to be a federated information systems manager, it was designed to be a general purpose directory. The query and search mechanisms of X.500, while fairly powerful, are much more limited than contemporary database systems.

X.500 has no standard means of defining and implementing behavior. Information systems like Orecom, Myriad, Interbase, and others all have embeded programing languages which allow the capabilities of the system to be extended.

6.2.3 ISO OSI

X.500 and X.400 are the flag ships of ISO OSI technology. Designed to bring forth real applications to the growing international computer networks, X.500 and X.400 have as their greatest liability their associations with ISO OSI technology.

ISO OSI technology has alway been controversial, especially in the United States. While initially challenged by the pundits of Internet technology, then politely tolerated, ISO OSI technology is mostly being ignored in terms of real Internet development. The reality of the evloving 'Information Superhighway' is that it is based on Internet technology, and most of that technology comes from the United States. The overwhelming feeling towards intoducing ISO OSI technology into the Internet is: "if it's not broken, don't try to fix it." By it's very association with ISO OSI, X.500 is at a serious disadvantage to new technologies which are 'Internet inventions.'

6.2.4 Complexity

ISO OSI critics are not without there valid critisisms. One of the biggest critisims is of the overwhelming complexity of the standards and technology. This complexity is also one of the biggest problems with X.500.

Complexity has made it difficult and expensive to develop X.500 as a technology, difficult to administer it, and often difficult to use. Development of the Internet Name Service took only a few short years since proposal of the standard to ubiquity of use. Little or no development of the INS is still done. Development of X.500 continues to this day and the cost of development continues to be quite high, primarily due to the complexity of the technology.

To the systems administrator, installing and maintaining an X.500 service is easily ten times more effort than the installing and maintaining the INS or Sun's NIS. This is directly related to the massive complexity of X.500 systems, especially with respect to schema controls.

The danger of using X.500 as a federated information systems manager is adding to this increasinly unmanagable complexity. By adding new features which X.500 was not designed for, there will be more working in setting up an maintaining the service, as well as increased likelyhood of malfuntions.

6.2.5 Obscurity

Although X.500 is an international standard which has been in existence since 1988, it has not overtaken similar systems like the Internet Name Service (INS) or Sun's Network Information Service (NIS). X.500 would make an excellent replacement for these two services, but has not. The Object Management Group (OMG) is defining new standards for distributed interoperability of objects and had developed its own name service. Services like Archie, Veronica, Jughead, Gopher, and the World-Wide-Web are based soley on the Internet Name Service.

Even though world-wide usage of X.500 continues to increase it has still not increased enough to the point where it considered for use in other newly evolving services. In spite of any technical merits X.500 may have, it is a failure in terms of marketing and consumerism. In these terms it is hard to justify development efforts based on a technology which is likely never become as ubiquitous as the INS or NIS.

6.3 Future Research and Development Needed

If this work is to continue successfully, there are a number of next steps possible. It is hard to offer a priority to such work as there are many goals to consider, but the most pragmatic course would be to continue establishing a working prototype with increasing functionality.

6.3.1 Add Support for Oracle

While the current prototype supports access to Sybase databases, the proxy support has been designed to easily implement other information providers. In particular, the SQL interface to both Sybase and Oracle would likely make this very easy as Sybase, Oracle, and other RDBS proxies would be implemented as subclasses of a more generic RDBS proxy class.

6.3.2 Complete X.500 Operations on RDBS Access

The current prototype supports the X.500 List and Read operations, but does not support Create, Delete, Modify, Modify RDN, Compare, or Search. Of these, Search is likely to be the most interesting and most useful as this is one of X.500's most powerful features, as well as one of the most powerful features of relational databases.

6.3.3 Support Hierarchy Mapping

While hierarchy mapping has been discussed at length in this thesis, no actual implementation exists. While such implementation is likely to be straightforward, real implementation experience often contradicts such assumptions.

6.3.4 Support Dynamic Registration

The current implementation allows the Directory Administrator or User to register an external information source with the directory via the Directory Access Protocol (DAP) and a standard Directory User Agent (DUA). However, it is necessary to restart the Directory Service Agent (DSA) before the new external information can be seen. It would be a very powerful ability to see the external information immediately after registration, or after changes to the registration information. This would promote an evolutionary approach to the registration process where the registration would proceed by successive refinements. It would also promote better information exploration for people either looking at new information sources, or looking at existing information sources in different ways (i.e., different hierarchy mappings).

6.3.5 Wide-Area Information Access

The current success of initiatives like the ftp servers, Gopher, and the World-Wide-Web, and applications such as Archie, Veronica, Jughead, and Mosaic demonstrate a clear desire for public access to world-wide information sources. However, these information systems are based on very simplistic information models which limit the ability of the user to control their view of the information.

By implementing some of the techniques of multidimensional indexing described in Section 5.3, experience could be gained as to the real benefits of a higher level wide-area information access.

6.3.6 Schema Translation Techniques

While X.500 supports an object-oriented information model, one aspect which sets X.500 apart from contemporary OODB systems is the lack of methods in X.500 (a means of implementing behavior). This should not be seen as a deficiency in X.500, but rather an opportunity to extend the X.500 standard with the latest technology in database access and interoperability mechanisms.

In particular, it would be very tempting to use techniques such as SchemaLog [29] and other techniques from Deductive Databases, Logic Programming Languages, Constraint Logic Programming, and Intelligent Systems research to define powerful mechanisms which could be used to register access and translation methods as entries within the X.500 Directory.

6.3.7 New Information Models

Perhaps X.500 is not the best information model for a canonical heterogeneous information model. Perhaps we can define a more powerful, more suitable model to federate heterogeneous information sources. However, we need not throw away X.500, rather we can adapt it to this model.

6.4 Final Remarks

Our ability to define and record information, to pass it on to others in a compact and efficient form, to pass it on permanently from one generation to the next is something shared by no other species. This information awareness is in a sense central to being a modern human in that we likely devote more resources to information and information management than to the basic needs of food, housing, and procreation.

Another aspect of the human condition is individuality; the need to express ideas in one's own terms, to set one's self apart from the rest of humanity. However, society pragmatically requires a degree of conformity as unchecked individuality can lead to inefficiency in social and economic mechanics. We see these tensions in the field of information systems between leading edge researchers and developers breaking new ground in information systems design and information system users who can ultimately only deal with so much diversity.

Also true to human nature is the need for interesting challenges. This is often truest in leading edge research laboratories where academics concentrate on the really tough problems, and leave the details or pragmatics to others for consideration. Unfortunately this often means that some important problems such as system usability or system administration and maintenance can be overlooked and not worked into a total solution.

Throughout this thesis there have been a couple of philosophical goals which were followed:

1. To not invent new technology unless necessary. Rather, retrofitting existing technologies such as X.500 formed the basis of this work.
2. To consider both the end-user of the technology as well as the administrator of the technology. Consequently, more emphasis was spent on issues such as registration than is typical in other research.

In particular, it is hoped that restricting the introduction of new technology and consideration of users and administrators can also be seen as interesting challenges.

Perhaps the biggest surprise to come out of this work was to see how issues of information systems federations and interoperability could be extended to improve on other emerging new technologies. In particular, the ability of a user or administrator to easily define new information views, such as custom designed information hierarchies, helps to build a bridge between individuality and conformity, giving us the best of both styles.

AppendixA X.500 Overview

X.500 is an international standard for a world-wide distributed directory system. Initial motivation for the design of the standard came in part from experience with X.400, the international standard for electronic mail, where a need was seen for some sort of directory service to map names of people into electronic mail addresses.

It was soon determined that there are many more uses for such a wide-spread directory than just e-mail users; for example, finding postal address, phone numbers, and other information related to specific people. It was also envisioned that the Directory could be used to map names into OSI network addresses (which are far more complex than Internet Protocol addresses) for systems and services. In short, it was decided that the Directory should be capable of storing information on just about anything.

A.1 Information Model

Information is stored in the directory as a collection of directory entries. Each entry in the directory is a collection of Attribute-Value Associations (AVA) which are of the form *attribute-type=attribute-value*. For example, the author might have an entry like:

```
commonName=Eric Kolotyluk,
                objectClass=sfuPerson,
                surname=Kolotyluk,
                userClass=staff,
                rfc822Mailbox=eric@cs.sfu.ca,
                otherMailbox=Eric_Kolotyluk@sfu.ca,
                otherMailbox=eric@sfu.ca,
                phoneNumber=+1-604-291-3014
```

These AVAs are basically information about an object the entry represents. It is interesting to note that multiple AVAs may share the same type, as in the case of the attribute-type otherMailbox.

Each entry in the directory is named by a collection of AVAs known as the Relative Distinguished Name (RDN). In the previous example, the RDN for the entry is the AVA commonName=Eric Kolotyluk, but it could just as well have been chosen to be two AVAs such as commonName=Eric and surname=Kolotyluk.

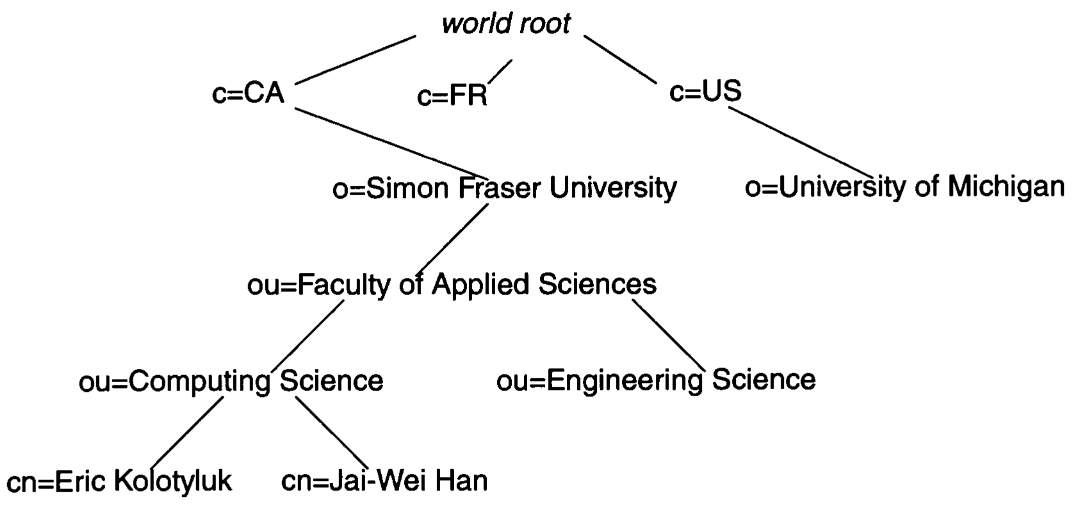
A series of Relative Distinguished Names is used to form a Distinguished Name (DN) for an entry. This DN gives each entry a name that is unique in the entire world-wide directory and can be thought of (in relational database terms) as the primary key of the entry. For example, the Distinguished Name of the author in the Directory is:

```
country=CA
organization=Simon Fraser University
organizationalUnit=Faculty of Applied Sciences
organizationalUnit=Computing Science
commonName=Eric Kolotyluk
```

where each AVA represents the RDN of a specific entry in the directory. Note: in most cases abbreviations are used for attribute-types and RDNs are grouped to form a string like: c=CA@o=Simon Fraser University@ou=Faculty of Applied Sciences@ou=Computing Science@cn=Eric Kolotyluk

As this can still be quite lengthy to type, a mechanism known as User Friendly Names (UFN) has been developed that offers shorter, yet possibly ambiguous names such as "Eric Kolotyluk, Simon Fraser University, CA" and some interfaces to the directory support aliases that support even shorter names like "Eric Kolotyluk, SFU, CA." Overall the intent is that the Directory user interface be as easy to use as possible, in spite of the complexity of the underlying mechanisms.

For convenience, directory entries are organized into a hierarchy known as the Directory Information Tree (DIT). A portion of the current tree looks like:



Note: there is no actual entry for the *world root*.

It is important to realize that each level of the DIT is represented by some entry with its own collection of attributes, and that the tree can be arbitrarily deep. In some directory implementations the attributes of an entry can even affect the behavior of subordinate entries (i.e., attribute inheritance).

The basic operations that can be performed on the Directory are:

Table 21: Basic Directory Operations

list	List the entries immediately subordinate to a particular entry.
read	Read the attributes of a particular entry.
add	Add a new entry subordinate to a given entry.
delete	Delete an entry from the Directory.
modify	Modify the attributes of an existing entry.
modify-RDN	Modify the Relative Distinguished Name of a given entry. This is a separate operation because unlike other attributes, the RDN must be unique among all sibling entries.
search	Search the directory for an entry or entries whose attributes match a given search expression.
compare	Compare the specified attributes of an entry with given values and indicate the results of the comparison. This is typically used to test a password match without allowing someone to read the password attribute.

The Directory is usually accessed in two ways:

1. Browsing via the *list* and *read* operations is intuitively very simple for most people.

2. Searching via the *search* and *read* operations is more powerful in that advantages can be taken of the parameterization of information contained in each entry via attributes. A search can compare the values of attributes for precise or imprecise matches. The latter is useful when searching for names when the exact spelling is not known and techniques like 'soundex' can be used. Also, unequal matches such as not-equal, greater-than, less-than, etc., can be specified. Finally, match specifications can be grouped with "and/or" to form powerful search patterns.

A.2 Schema Model

The Directory has a very strong sense of schema in that there are many rules that govern which attributes an entry may hold, and what values attributes may take. In some cases (i.e., quipu) there are even rules about the structure and contents of specific subtrees.

A.2.1 Object Class

The most obvious schema control in the Directory is the requirement that each entry belongs to a particular object class. This not only categorizes the entry, but determines which attributes the entry must have and/or may have. There is a specific attribute-type known as objectClass which every entry in the directory must have. Every entry in the directory implicitly belongs to the object class top, which mandates the presence of the objectClass attribute.

X.500 specifies a set of predefined object classes. For example, if an entry has an AVA of objectClass=person, then the entry must also have attributes commonName and surname, and may optionally have any of the attributes: telephoneNumber, seeAlso, description, and userPassword.

It is also possible to define an object class as an extension or a subclass of another object class. For example, the predefined X.500 object class residentialPerson is subclass of the object class person and inherits the same conditions with respect to the attributes commonName, surname, telephoneNumber, seeAlso, description, and userPassword. In addition, the object class residentialPerson requires the entry to have a localityName attribute, and any of the attributes: streetAddress, postalAddress, postalCode, stateOr-ProvinceName, postOfficeBox, physicalDeliveryOfficeName, preferredDeliveryMethod, facsimileTelephoneNumber, internationaliSDNNumber, teletexTerminalIdentifier, telex-Number, preferredDeliveryMethod, destinationIndicator, registeredAddress, x121Address, or businessCategory.

It is also possible for an entry to belong to more than one object class. This is denoted by having more than one objectClass AVA.

A.2.2 Syntax

The directory also governs the syntax of attribute values based on attribute-types. For example, attributes such as commonName, surname, and description have a syntax of CaseIgnoreString whereas the serialNumber attribute has a syntax of PrintableString.

The syntax of attribute values also affects how search and compare operations proceed. For example, the CaseIgnoreString syntax will match two values regardless of the chosen case of the alphabetic characters, whereas PrintableString requires an exact match. This is actually an oversimplification of the X.500 standard, but is sufficient for this presentation.

A.2.3 Inherited Attributes

Though not part of the X.500-1984 standard, the quipu implementation of X.500 supports the use of inherited attributes. Basically the inheritedAttribute attribute specifies a set of AVAs that are to be inherited by the immediate subordinate entries.

A.2.4 Tree Structure

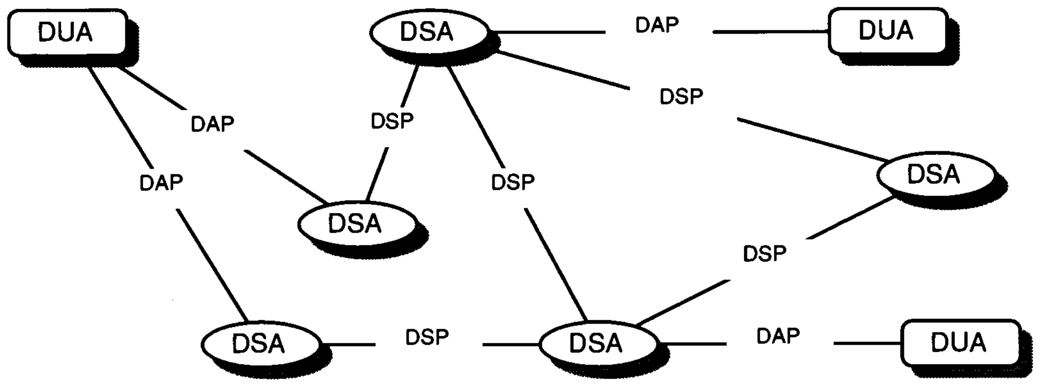
Though not part of the X.500-1984 standard, the quipu implementation of X.500 introduces the treeStructure attribute. This attribute specifies which object classes are permitted for all entries subordinate to the entry having the treeStructure attribute.

A.3 Operational Model

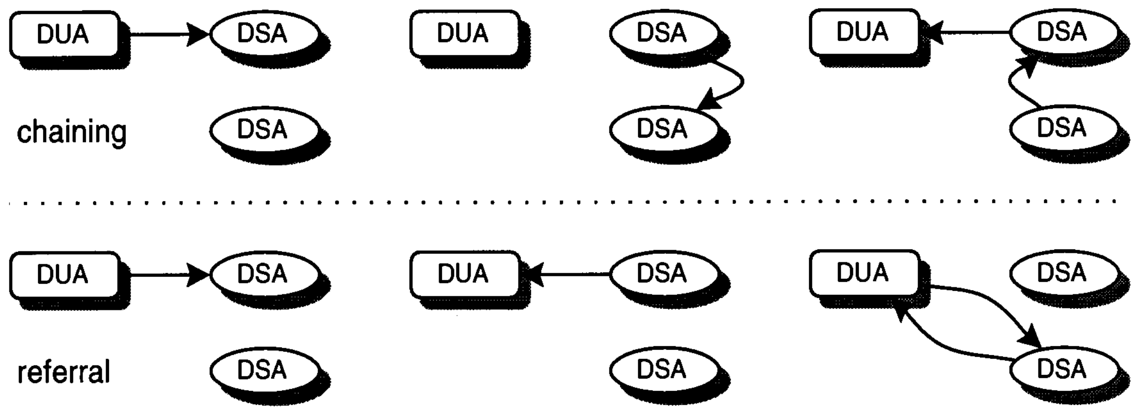
The operational model of the Directory is not of much importance to directory users as all they should be aware of is that the directory is a service. To the directory administrator and directory implementor the operational model is extremely important.

The two most important aspects of the operational model are the Directory User Agent (DUA) and the Directory Service Agent (DSA). Basically the directory service is implemented by a number of cooperating DSAs each managing a portion of the Directory Information Base (DIB). In this way the directory becomes a truly distributed application. This distribution is key to the Directory's scalability world-wide.

DSAs communicate with one another via the Directory Service Protocol (DSP) and DUAs communicate with other DSAs via the Directory Access Protocol (DAP). DUAs never communicate with other DUAs.



Because the Directory is distributed, often one DSA cannot satisfy a request from a DUA. There are two mechanisms available for DSAs to satisfy service requests from DUAs. Through ***chaining*** a DSA can pass on a request to one or more other DSAs. When the result are received they can be combined with other results, including those from the DSA's own DIB, and passed back to the DUA. Through ***referrals*** the DSA can decline to service the complete request, but indicate to the DUA one or more DSAs that likely can service the request.



For a given directory request, it is possible that some arbitrary amount of chaining and referrals will proceed.

A.4 Security Model

One of the most important features of the OSI Security Model is the use of RSA public key encryption. To make use of public key encryption it is essential that the *public* keys be available in some public place—the Directory is most obvious place for this.

A.4.1 Authentication

However, the Directory is not merely a repository for public keys. It is also a user of the mechanism in that it supports the notion of Authentication through ***binding*** to the directory. In the simplest case anyone (or anything) may attempt to bind to the directory anonymously via some DSA, but the DSA may refuse this according to local policy. Normally, however, a DUA will attempt to bind a user by identifying the user to the directory. In this case, the user will have an entry in the directory, typically containing a password attribute which the DUA will perform a compare request against. Under more stringent circumstances the Directory can require/provide strong authentication in which the DSA being bound to challenges the DUA with a random signature encrypted with the user's public key. The DUA must decrypt the signature, then encrypt it again with the user's *private* key, ensuring the user is not a pretender.

A.4.2 Access

Though not yet part of the standard [I think], the Quipu implementation of X.500 makes use of access control lists to control access to an entry, to its subordinate entries, and even to each individual attribute.

As countries like Canada and the United States increasingly create legislation governing individual rights to information—the right to know what information an organization holds about you, the right to have erroneous information corrected, and even the right to what information an organization can share with others—increasingly it will be important that organizations can abide by these laws. For example, if you went to an organization and asked to see all the information they kept on you, the organization might be hard pressed to tell you. If you asked that your home phone number not be shared with anyone outside the organization, likely their computer database systems could not implement such a request easily. If many people came to the organization with similar requests, the administrative, clerical, and technical staff could be easily overwhelmed trying to satisfy such requests, and even become unable to comply even though the law requires it.

By using mechanisms like those found in QUIPU, we could give people increased access to information about themselves, and increased control over how the information is used (who can see it, who can change it).

A.4.3 Dredging

Even with authentication and access controls, it is still possible for unscrupulous individuals to collect information from the directory through normal channels, yet violate the intended used of the information via statistical or other analysis. For example, by making repeated compare attempts they may try to guess a password, or by repeated read requests they may attempt to gather a profile on an organization. QUIPU guards against this somewhat by allowing the DSA administrator to cap the number of requests made by a specific individual.

A.5 ISODE

The ISO Development Environment is a project started in the mid 1980's by Marshal T. Rose as an attempt to create an easily accessible test-bed for the implementation of ISO OSI distributed applications. Central to this project was the development of a working ISO protocol stack. More important, however, was the development and distribution of tools for constructing ISO OSI applications which enabled development laboratories around the world to contribute to the overall development effort.

For the most part, the ISODE distribution exists in the 'public domain' in that anyone is free to receive and use a copy of the distribution. Indeed hundreds, possibly thousands, of sites around the world have installed ISODE and its OSI applications.

A.5.1 Quipu

In the ancient Inca civilization information was recorded on knotted strings called 'quipus.' These quipus were central to the functioning of a village in that they told who lived where and owned what land, what debts were owed, etc.

By the late 1980's University College London (UCL) embarked on a project to implement the ISO OSI Directory service via ISODE, and chose the name Quipu to symbolize the importance to the global information village.

A.5.2 GDA

The Generic Directory API[1] is a well defined interface between the QUIPU DSA and the underlying persistent data store. It was designed to make it easier for developers to implement new storage systems or adapt QUIPU to existing ones. The results of this thesis indicate that GDA is a good start, but that there are important enhancements that would benefit the design.

A.5.3 Consortium

In 1992 it was decided that the ISODE effort should be more formally recognized and funded and so the ISODE Consortium (IC) was formed. The IC continues to develope and improve the on ISODE Version 8 (the last publicly available version), but makes the development environment easily accessible to academics at a small cost.

[1] Application Program Interface

AppendixB Sample Directory Session

The current implementation of the heterogeneous database proxy support is based on ISODE Consortium release 1.2 of Quipu and the Generic Directory API (GDA). At this time a special version of the qb disk database has been modified to support the proxy model as this was expedient in terms of implementation strategy. A more effective design would be to incorporate the proxy support into the main body of quipu, or to significantly enhance the current GDA component.

What this appendix demonstrates is a sample session with Quipu's dish (DIrectory SHell) and Sybase's isql (Interactive SQL). Note, the following specimen output has been edited for readability.

```
aquarius{eric}1: isql
Password:
1> sp_help Part
2> go
 Name                           Owner                          Type
 ------------------------------ ------------------------------ ----------------
 Part                           dbo                            user table

 Data_located_on_segment        When_created
 ------------------------------ -------------------------
 default                               Mar  2 1994 11:47AM

 Column_name     Type            Length Nulls Default_name    Rule_name
 --------------- --------------- ------ ----- --------------- ---------------
 number          int                  4     0 NULL            NULL
 name            varchar             16     0 NULL            NULL
 price           smallmoney           4     0 NULL            NULL
 description     varchar             64     0 NULL            NULL
Object does not have any indexes.
No defined keys for this object.

(return status = 0)
1> 1> select * from Part
2> go
 number  name        price      description
 ------ ----------- --------   ------------------------------------------------
  12345 HD125s        259.00   125 MB SCSI hard disk
  12346 HD233sf       427.00   233 MB fast SCSI hard disk
  12347 HD455sf       721.00   455 MB fast SCSI hard disk
  12348 HD785sf     1,033.00   785 MB fast SCSI hard disk
  12349 HD785sfd    1,121.00   785 MB 16-bit fast SCSI hard disk

(5 rows affected)
1>
```

Here we see an example of a parts-price list a small company might want to publish via the Directory.

Using the Directory User Agent (DUA) for Quipu we list the entries under the entry for the Open Systems Laboratory. The first 9 entries are stored in the GDA-QB directory information base via GDBM files. The last 5 entries are listed as a side effect of the existence of entry 9, proxy=Parts List, and are retrieved from the specified Sybase server.

```
aquarius{eric}4: dish
Welcome to Dish (DIrectory SHell)
Dish -> moveto osl
c=CA@o=Simon Fraser University@ou=Faculty of Applied Science@ou=Computing
Science@ou=Open Systems Laboratory
Dish -> list
1   commonName=aquarius
2   commonName=Directory Developer
3   commonName=Jane Doe
4   commonName=Joe Blow
5   commonName=John Doe
6   commonName=David Doe
7   surname=Kolotyluk%commonName=Eric
8   commonName=Max Knife
9   proxy=Parts List
10  commonName=HD125s
11  commonName=HD233sf
12  commonName=HD455sf
13  commonName=HD785sf
14  commonName=HD785sfd
Dish -> showentry 9
c=CA@o=Simon Fraser University@ou=Faculty of Applied Science@ou=Computing
Science@ou=Open Systems Laboratory@proxy=Parts List
objectClass            - quipuProxySybase
lastModifiedTime       - Wed Mar 30 17:58:11 1994
lastModifiedBy         - countryName=CA
                         organizationName=Simon Fraser University
                         organizationalUnitName=Faculty of Applied Science
                         organizationalUnitName=Computing Science
                         commonName=Eric Kolotyluk
accessControlList      - others can read the child
                         self can write the child
                         others can read the entry
                         self can write the entry
                         others can read the default
                         self can write the default
proxy                  - Parts List
proxiedObjectClass     - sfuPart=Part
proxiedAttribute       - partNumber=number
proxiedAttribute       - description=description
proxiedAttribute       - commonName=name
proxiedAttribute       - caPrice=price
proxiedName            - commonName
sybaseServer           - CMPT
sybaseUser             - eric
sybaseUserPassword     - Read but not displayed
sybaseApplication      - QUIPU Proxy
sybaseDatabase         - eric
Dish -> showentry 10
c=CA@o=Simon Fraser University@ou=Faculty of Applied Science@ou=Computing
Science@ou=Open Systems Laboratory@cn=HD125s
objectClass            - sfuPart
commonName             - HD125s
description            - 125 MB SCSI hard disk
partNumber             - 12345
caPrice                - $259.00
Dish ->
```

In this session note the attributes `proxiedObjectClass`, `proxiedAttribute`, and `proxiedName` are used to specify the mapping from the relational schema to the directory schema. The `proxiedObjectClass` specifies the table name to construct entries from, as well as the directory object class the proxied entries will belong to. The `proxiedAttribute` specifies each relational attribute name and the directory attribute these are mapped to. Finally, the `proxiedName` specifies which attributes form the Relative Distinguished Name (RDN) of each proxied entry.

The other significant attributes, `sybaseServer`, `sybaseUser`, `sybaseUserPassword`, `sybaseApplication`, and `sybaseDatabase` are configuration information for the proxy mechanism to connect to the appropriate server.

Finally, we see the effect of reading one of the proxied entries, `HD125s`, and note that this corresponds to the relational database. If we alter the underlying relational database,

```
1> insert Part values(23456, "HD1240sf", $1389.00, "1.2 GB fast SCSI hard disk")
2> go
(1 row affected)
1>
```

we can immediately see the result via the Directory.

```
Dish -> showentry commonName=HD1240sf
c=CA@o=Simon Fraser University@ou=Faculty of Applied Science@ou=Computing
Science@ou=Open Systems Laboratory@cn=HD1240sf
objectClass          - sfuPart
commonName           - HD1240sf
description          - 1.2 GB fast SCSI hard disk
partNumber           - 23456
caPrice              - $1389.00
Dish ->
```

The main significance of this is that once the DSA has been configured, any changes in the underlying relational database are automatically available via the Directory. This is an enormous saving for any database or directory administrators who might otherwise have to develop and maintain custom mechanisms to download data from the relational database to the directory, as well as ensure that this happens on a timely basis. Using proxies, the directory administrator merely creates a proxy entry with the appropriate set of attributes, then restarts the DSA.

Currently the proxy's configuration information is used only when the DSA starts up, so the proxy entry must exist prior to DSA initialization. This is just a matter of development expedience however. The ultimate intent is to have the proxy support become active the moment the proxy entry is created. The main reason for not implementing this feature is that it is still necessary, typically, for the directory administrator to modify the underlying Quipu object-identifier (OID) tables, then restart the DSA anyway.

Prior to the above session, the author had to create the following entries in the Quipu oidtable files: the attribute types `partNumber` and `caPrice`, and the object class `sfuPart`. The attributes `commonName` and `description` already existed and could be just as easily used or not. For example, the author might have decided to create a new attribute type, `partName`, and used it instead of `commonName`.

It is hoped that after Quipu is converted to X.500-1993, object-identifiers can be managed through the Directory itself. This would also lead the way for automatic creation of new object identifiers and object classes based on the configuration parameters of the proxy entry. Ultimately, registering a relational database (or any other information source) with the Directory should be a simple, straightforward, *one-step process* for a systems administrator, or even an ordinary computer user.

References

[1] Marshall T. Rose. *The Open Book: A Practical Perspective on OSI.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[2] CCITT. The Directory—Overview of Concepts, Models and Services, CCITT X.500 Series Recommendations. CCITT, December 1988.

[3] Gerald Neufeld, Barry Brachman, Murray Goldberg, Duncan Stickings, *The EAN X.500 Directory Service,* Internetworking: Research and Experience, Vol. 3, pp. 55-81, John Wiley & Sons, 1992.

[4] Daniel L. Silver, James W. Hong and Michael A. Bauer, *X.500 Directory Schema Management,* International Conference on Data Engineering, Houston, Feb. 1994, pp. 393-401.

[5] OMG, *The Common Object Request Broker: Architecture and Specification,*

[6] M Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, D.C. Steere, *Coda: a Highly Available File System for a Distributed Workstation Environment,* IEEE Transactions on Computers 39(4), April, 1990, pp. 447-459.

[7] M.N. Nelson, B.B. Welch, J.K. Ousterhout, *Caching in the Sprite Network File System,* ACM Transactions on Computer Systems 6(1), February, 1988, pp. 134-154.

[8] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, B.B. Welch, The Sprite Network Operating System, Computer 21(2), February, 1988, pp. 23-36.

[9] B. Walker, J. Popek, R. English, C. Kline, G. Thiel, *The LOCUS distributed operating system,* ACM SIGOPS, Oper. Syst. Rev. 17, 5 (Oct.) 1983, pp. 49-70.

[10] Henry F. Korth and Abraham Silberschatz, *Database System Concepts, Second Edition,* (1991) McGraw-Hill, Inc., New York, ISBN 0-07-044754-3.

[11] Jeffrey D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies,* 1989, Computer Science Press, Rockville, MD 20850, ISBN: 0-7167-8162-X.

[12] A. Makinouchi, *A Consideration of Normal Forms on Not Necessarily Normalized Relations in the Relational Data Model,* Proceedings of the International Conference on Very Large Data Bases (1977), pages 447-453.

[13] G. Jaeschke and H. J. Schek, *Remarks on the Algebra of Non First Normal Form Relations,* Proceedings fo the ACM SIGACT-SIGMOD Symposium on Priciples of Database Systems (1982), pages 124-138.

[14] C. R. Carlson and R. S. Kaplan, A Generalized Access Path Model and Its Application to Relational Database System, Proc. 1976 ACM SIGMOD International Conference on Management of Data, Washington, D. C. (June 1976).

[15] *ISO-ANSI Working Draft, Database Language SQL (SQL3),* February 5, 1993, Digital Equipment Corporation, Maynard, Massachusetts.

[16] ACM Computing Surveys: Volume 22, Number 3, September 1990 (ISSN 0360-0300) Association for Computing Machinery, New York, NY 10036.

[17] Ahmed K. Elmagarmid, Calton Pu, *Guest Editor's Introduction to the Special Issue on Heterogeneous Databases*, [16], pp. 175-178.

[18] Amit P. Sheth, James A. Larson, *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases*, [16], pp. 183-236.

[19] Gomer Thomas, Glenn R. Thompson, Chin-Wan Chung, Edward Barkmeyer, Fred Carter, Marjorie Templeton, Stephen Fox, Berl Hartman, *Heterogeneous Distributed Database Systems for Production Use*, [16], pp. 237-266.

[20] Witold Litwin, Leo Mark, Nick Roussopoulos, *Interoperability of Multiple Autonomous Databases*, [16], pp. 267-293.

[21] Research Issues in Data Engineering: Interoperability in Multidatabase Systems, Proceedings April 1993, IEEE Computer Society Press, IEEE Computer Society Press, Los Alamitos, California, USA.

[22] Daniel A. Keim, Hans-Peter Kriegel, Andreas Miethsam, *Integration of Relational Databases in a Multidatabase Systems based on Schema Enrichment*, [21], pp. 96-104.

[23] Aidong Zhang and Jin Jing, On Structural Features of Global Transactions in Multidatabase Systems, [21], pp. 199-206.

[24] Hongjun Lu, Beng-Chin Ooi, Chen-Hian Goh, *Multidatabase Query Optimization: Issues and Solutions*, [21], pp. 137-143.

[25] Ninth International Conference on Data Engineering, Proceedings, April 1993, IEEE Computer Society Press, IEEE Computer Society Press, Los Alamitos, California, USA.

[26] L. Suardi, M. Rusinkiewicz, W. Litwin, Execution of Extended Multidatabase SQL, [24], pp. 641-650.

[27] Ahmed K. Elmagarmid (ake@cs.purdue.edu), Jiansan Chen (jchen@cs.purdue.edu), Weimin Du (du@hpl.hp.com), Rob Pezzoli (robp@bnr.ca), Omran Bukhres (bukhres@mhd1.moorhead.musu.edu), *InterBase: An Execution Environment for Global Applications over Distributed, Autonomous, and Heterogeneous Software Systems*, Technical Report.

[28] Xiangning Liu (xl@cs.purdue.edu), Jiansan Chen (jchen@cs.purdue.edu), Rob Pezzoli (robp@bnr.ca), *The InterBase View Graphical User Interface*, Technical Report.

[29] Laks V. S. Laksmanan, Fereidoon Sadri, Iyer N. Subramanian, *On the Logical Foundations of Schema Integration and Evolution in Heterogeneous Database Systems*, DOOD'93, e-mail: {laks, sadri, subbu}@cs.concordia.ca

[30] Christiaan Thieme (ct@cwi.nl) and Arno Siebes (arno@cwi.nl), *An Approach to Schema Integration Based on Transformations and Behavior*, Technical Report, CWI, P.O. Box 4079 AB Amsterdam, The Netherlands

[31] S. Y. W. Su, S. C. Fang, H. Lam, *An Object-Oriented Rule-Based Approach to Data Model and Schema Translation*, Technical Report. Database Systems Research and Development Center, CSE#470, Department of Computer and Information Sciences, Department of Electrical Engineering, University of Florida, Ganiesville, FL 32611, e-mail: su@pacer.cis.ufl.edu, {sf, hlam}@reef.cis.ufl.edu

[32] S. Y. W. Su, S. C. Fang, A Neutral Semantic Representation for Data Model and Schema Translation, Technical Report Number: TR-93-023, Database Systems Research and Development Center, CSE#470, Department of Computer and Information Sciences, Department of Electrical Engineering, University of Florida, Ganiesville, FL 32611, e-mail: su@pacer.cis.ufl.edu, sf@reef.cis.ufl.edu

[33] S. Chakravarthy, K. Karlapalem, S. B. Navathe, A. Tanaka, *Database Supported Cooperative Problem Solving,* Technical Report UF-CIS-TR-92-046, Department of Computer and Information Sciences, Computer Science Engineering Building, University of Florida, Gainesville, Fl 32611, e-mail: sharma@snapper.cis.ufl.edu

[34] W. K. Whang, S. Chakravarthy, S. B. Navathe, *Heterogeneous Databases: Inferring Relationships for Merging Component Schemas, and Query Language,* Technical Report UF-CIS-TR-92-048, Department of Computer and Information Sciences, Computer Science Engineering Building, University of Florida, Gainesville, Fl 32611, e-mail: sharma@snapper.cis.ufl.edu

[35] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, F. Lambay, *A Federated Multi-media DBMS for Medical Research: Architecture and Functionality,* Technical Report UF-CIS-TR-93-006, Department of Computer and Information Sciences, Computer Science Engineering Building, University of Florida, Gainesville, Fl 32611, e-mail: sharma@snapper.cis.ufl.edu

[36] D. Clements, M. Ganesh, S.-Y. Hwang, E.-P. Lim, K. Mediratta, J. Srivastava, J. Stenoien, H.-R. Yang, *Myriad: Design and Implementation of a Federated Database Prototype,* Technical Report, Deparment of Computer Science, University of Minnesota, Minneapolis, MN 55455.

[37] Ee-Peng Lim, Jaideep Srivastava, *Entity Identification in Database Integration: An Evidential Resasoning Approach,* University of Minnesota, Minneapolis, MN 55455.

[38] Sharad Mehrotra, Henry F. Korth, Avi Silberschatz, *An Architecture for Large Multidatabase Systems,* Technical Report, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-1188.

[39] Yoram Kornatzhy, Peretz Shoval, Reverse Engineering from Relational to Object-Oriented Databases, LAKS, October 4, 1993

[40] Pattie Maes, *Concepts and Experiments in Computational Reflection,* OOPSLA Proceedings, ACM SIGPLAN Notices, Vol. 22, October 1987, pp. 147-155.

[41] Brian C, Smith, *Reflection and Semantics in LISP,* Conference record of the ACM Symposium on Principles of Programming Languages, pp. 23-35, ACM Press, 1984.

[42] Akinori Yonezawa, Takua Watanabe, *An Introduction to Object-Based Refective Concurrent Computations,* Proceedings of the 1998 ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Vol. 24, pp. 50-54, SIGPLAN Notices, April 1989.

[43] Hidehiko Masuhara, Satoshi Matsuoka, Takuya Watanabe, Akinori Yonezawa, *Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently,* OOPSLA Proceedings, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, pp. 127-144.

[44] Yasuhiko Yokote, *The Apertos Reflective Operating System: The Concept and its Implementation*, OOPSLA Proceedings, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, pp. 414-434.

[45] Jia-Wei Han and Yongjian Fu, *Dynamic Generation and Refinement of Concept Hierarchies for Knowledge Discovery in Databases*, AAAI '94 Workshop on Knowledge Discovery in Databases, Seattle, Washington, July 1994, pp. 157-168.

[46] Ayellet Tal (ayt@princeton.edu) and Rafael Alonso (alonso@mitl.com), *Integration of Commit Protocols in Heterogeneous Databases*, Technical Report, September 1992.

[47] Nandit Soparkar, Henry F. Korth, Abraham Silberschatz, *Techniques for Failure-Resilient Transaction Management in Multidatabases*, Technical Report TR-91-10, Department of Computer Sciences, University of Texas at Austin, December 1991.

[48] Sharad Mehrotra, Rajeev Rastogi, Yuri Breitbart, Henry F. Korth, and Abraham Silberschatz, *The Concurrency Control Problem in Multidatbases: Characteristics and Solutions*, Technical Report TR-91-37, Department of Computer Sciences, University of Texas at Austin, December 1991.

[49] Nandit Soparkar, Henry F. Korth, Abraham Silberschatz, *Transaction Managment for Distributed Mutlidatabases*, Technical Report TR-92-18, Department of Computer Sciences, University of Texas at Austin, December 1991.