1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

Implementing LDAP Directory Hierarchy with Relations

**Shepherd S.B.Shi, **C.S.Yang and *David Lin*

*Distributed System Services, IBM Austin
11400 Burnet Rd. , Austin, TX 78758
**Institute of Computer and Information Engineering
National Sun Yat-Sen Unversity
Kaohsiung, Taiwan, R.O.C.

Abstract

LDAP, is one of the emerging technology which can provide directory services to applications ranging from email systems to distributed system management tools. However, the reference implementation of LDAP from University of Michigan needs a lot of enhancements for being a reliable and Scaleable enterprise directory service. This paper presents a design and implementation to use a relational database as a robust, scaleable data store for LDAP directory information. One of the sailent feature of our solution is to be able to implement LDAP hierarchies with relational tables efficiently.

1. Introduction

Directory services, a critical part of distributed computing, is the central point where network services, security services and applications can form a integrated distributed computing environment. The current usage of a directory service can be classified into the following categories:

Name Service - Use directory as a source to locate internet host address or the location of the server. For example, DNS and DCE CDS.

User registry - To store information of all users in a system. Especially if the system is composed of a number of interconnected machine, a central repository of user information will enable the system administrator to administer the distributed system as a single system image. Novel's NDS, is an example.

Yellow page lookup - Some modern email clients provide users with the capability of looking up people's names and email addresses. The users typed in name, or part of the name and the directory service will extract the email information for the user. Netscape Communicator, Lotus Notes, Endora and other email clients provide the address book look up capability.

With more and more applications and system services demanding a central information repository, the next gener ation directory service will be providing system administrators with a data repository which could significantly ease the administrative burden. In addition, the future directory service will also provide end users with a rich information data warehouse which allows them to access department or company employee data, or resource information such as name and location of printers, copy machines, etc...
In the internet/intranet environment, users will be able to use the public key certificates constrained in the directory to handle encrypted or digitally signed documents.

LDAP, is one of the emerging technology which can provide directory services to applications ranging from email systems to distributed system management tools. LDAP is an open Internet standard, produced by the Integrated Engineering Task Force (IETF). LDAP provides the capability for directory information to be queried or updated. LDAP offers a rich set of searching capabilities with which users can put together complex queries to get desired information from the backing store.

LDAP is originally implemented by University of Michigan. The U. of M. reference implementation is freely available through the FTP site. The U. of M. LDAP is implemented based on several freely available btree packages, such as GNU dbm and Berkely db44 packages. This reference implementation supports version 2 LDAP protocol and is used as a basis for LDAP/DB2. LDAP Version 2 protocol is defined in IETF RFCs 1778-1779.

The U. of M. LDAP is a sound reference implementation for people to understand the internals of LDAP. However, it still needs a lot of enhancements for being a reliable and Scaleable enterprise directory service. The relational databases, such as DB2 or Oracle, could be the ideal backing store for LDAP directory. The relational databases provide hard-won advantages such as scaleability, transaction integrity, backup and recovery, stability and a powerful query processing engine. However, Mapping the LDAP model [5] to relational tables is not a trivial task. First of all, it appears that the LDAP model is hierarchical. It is known that hierarchies are very easy to represent in the hierarchical databases (like IMS) , where the structure of the data and the structure of the database are the same. Unfortunately, it is a general opinion that relational databases do not provide adequate support for such data. Second, LDAP allows both single and multi-valued attributes. But relational database does not deal with multi-valued attributes well. This paper presents a practical and efficient solution for the problems mentioned above. The design and implemenation mentioned in this paper is used in the IBM E-Directory LDAP server. Our performance results showed that our implementation is competitive to other directory products in the industry.

The rest of this paper is organized as follows: Section 2 descibes a summary of related work. Section 3 presents the LDAP directory model. Section 4 describes our design and implementation to map LDAP model to relational tables in detail. Section 5 reports some performance measurements. Section 6 is the conclusion.

2. Summary of Related Work

LDAP is originally implemented by University of Michigan. The U. of M. reference implementation [1] is freely available through the FTP site. The U. of M. LDAP is implemented based on several freely available b-tree packages, such as GNU dbm and Berkeley db packages. This reference implementation supports version 2 LDAP protocol and is used as a basis for LDAP/DB2. LDAP Version 2 protocol is defined in IETF RFCs 1777-1779 [3,6,7].

The U. of M. LDAP is a sound reference implementation for people to understand the internals of LDAP. However, it needs a lot of enhancements for being a reliable and scaleable enterprise directory service. One of the limitations is that it does not scale to more than a few hundred thousand to possibly a million entries. The use of simple file-system based hash and b-tree packages will not be able to handle large amounts of data. On the other hand, relational database technologies like DB2 [10] is designed to handle up to tera bytes of data. Second, populating directories with large numbers of entries is time consuming work. It takes a lot of time to populate the directories with millions of entries. Third, because of limited search and indexing facilities provided by the file-system based back-end, only candidate entries can be retrieved. Then each entry is filtered through the filter program before it is returned to the client. However, in some cases, when the set of candidates is large, the search degenerates into a sequential search. For example, we discovered that negation queries and existence queries are fairly expensive with both the reference implementation and LDAP server from Netscape. By using the powerful search engines provided by DB2, we were able to address some of the weak areas.

Mapping the LDAP model [5] to relational tables, however, is not a trivial task. First of all, LDAP allows both single and multi-valued attributes. But relational database does not deal with multi-valued attributes well. Second, the size of each DB2 table [10] is limited to 4K. Third, it appears that the LDAP model is hierarchical. It is known that hierarchies are very easy to represent in the hierarchical databases (like IMS) , where the structure of the data and the structure of the database are the same. Unfortunately, it is a general opinion that relational databases do not provide adequate support for such data. It does not directly map hierarchical data into tables because tables are based on sets rather than on graphs. Different vendors provide different mechanisms for the tree structure. For example, DB2 [10] provides the WITH clauses in the select statement to provide subtree traversal with arbitrary depth. Oracle [16] has CONNECT BY PRIOR and START WITH clauses in the SELECT statement to provide partial support for reachability and path enumeration. But all mechanisms will end up with recursive queries to handle hierarchical structures. Through experiments, we discovered that recursive queries do not scale up well for large number of records in the table. We did a small experiment with 1000 LDAP entries using DB2 recursive queries; a simple select takes more than five minutes to complete.

U.S. patent 5467471 [16] presented a solution which does not require recursive query. The invention provides a genealogy table with which the directory hierarchy is represented in a table form. Each column of the genealogy table represents a level of the directory tree. This solution might be fine for directories with limited hierarchy depth. However, it is very difficult to realize the idea in practice when the directory is infinitely deep and the number of columns of a table is limited. We have attempted a similar implementation but learned that the complexity is so great and the performance implication is unclear.

To address this problem, we invented an efficient method to represent LDAP hierarchies with relational tables without the overhead of recursive queries [11,12,13,14,15]. Our performance results showed that our implementation is competitive to other directory products in the industry.

3. LDAP Information Model

The LDAP directory database consists of entries. Each entry is composed of one more attributes. A type is associated with each attribute, and an attribute can have more than one value in an entry. The attribute type determines the syntax of the attribute. The syntax of an attribute determines how the data will be compared against the values in the query. In LDAP V2 [2,3,4,6,7], the possible syntaxes and their meanings are:

bin: binary
ces: case exact string
cis: case ignore string

tel: telephone number string (like cis but blanks and dashes are ignored during comparisons)
dn : distinguished name

Users can define the attributes which can be included in the entry in the LDAP schema. The set of attributes is named object classes. Mandatory attributes are required to have values in the entry. Optional attributes, on the other hand, do not have to exist for the entry of an object class. For example, to define an objectclass called Person, the following is a possible definition:

```
objectClass Person
required cn,sn,objectclass
allows   mail, phone, address, fax
```

In object class Person, cn, sn and objectclass are mandatory attributes; mail, phone, address and fax are optional attributes.

LDAP entries are arranged in a tree structure that follows a geographical and organizational distribution. Each entry is uniquely identified by a distinguished name (DN). The formal definition of a distinguished name (DN) is given in RFC 1779 [6].

The functions provided by LDAP can be categorized as:

- Query: Search and Compare. These operations are used to retrieve information from the database. For the search function, the criteria of the search is specified in the search filter. The search filter is a Boolean expression which consists of attribute name, attribute value, and Boolean operations like AND, OR and NOT. Users can use the filter to perform fairly complicated search operations. The filter syntax is defined in RFC 1960 [8].

  In addition to the search filter, users can also specify where the search starts in the directory tree structure. The starting point is called the based DN. The search can be applied to a single entry (based level search), an entry's children (one level search), or an entire subtree (sub tree search). Although LDAP does not support a head and list operation directly, the search operation is used to emulate these operations by setting the DN, scope, and filter appropriately.

- Update: Add, Delete Modify and Modify RDN. Users can use these functions to update the contents of the directory.

- Authentication. Bind and Unbind. LDAP supports simple id and password based authentication scheme. In the bind operation, user can specify the id and password and the server will use this information to authenticate the client. If successful, the authentication is in effect for the life of a LDAP session.

- Others.
  - MODRDN to rename an existing directory entry
  - INIT to initialize LDAP client library and obtain a session handle
  - RESULT to retrieve the results of the search operation

4. Model LDAP through Relations

At first glance, it seems very obvious that we should be mapping a LDAP object class into a DB2 relation. However, this mapping posed a serious problem since LDAP model allows both single and multi-valued attributes. In the database design guideline, the First Normal Form requires that attributes within each tuple are ordered and complete and that the domains permit only simple values. Simple values can not be decomposed into multiple values and cannot themselves be sets or relations. Some database systems (like DB2) are attempting to support multi-valued attributes. However, the implementation is not available yet. UN-normalized relations will make update operations (e.g.. add, modify and delete) fairly difficult to manage. We also discovered that we might lose some data semantics during the update process when multi-valued attributes exist.

Instead, we mapped each LDAP attribute, which can be searched by the user, to an attribute relation. This relation consists of two columns: unique identifier EID and normalized attribute value. In our system, each LDAP entry is assigned an EID. Based on the attribute syntax, the attributes are converted (or normalized) so that our system can apply SQL queries to the attribute values. For example, if the attribute syntax is case insensitive (CIS), the attribute value will be converted to all upper case and stored in the attribute table. The attribute table is used mainly for search operation to find the entries which match the filter criteria. The actual entry data, is stored in the LDAP_ENTRY table. In other words, the SQL queries generated by our system will use the attribute table to locate the entry EIDs which match the filter expression and use the EIDs to retrieve the entry data from the LDAP_entry table. Another advantage of this per attribute table is that the size of the entry is no longer bounded by the DB2 4k limit. The attribute table and ldap_entry table is similar to the id2entry and attribute indexes in the U. of M. reference implementation. The main difference is that our implementation is able to retrieve the exact target entries instead of just "candidates". As a result, no post-processing of filtering entries is needed in LDAP/DB2.

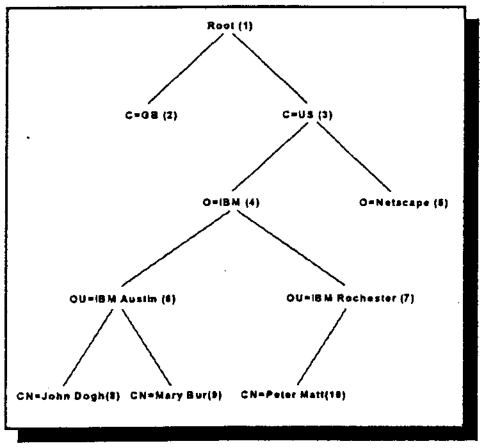
A second challenge is to map LDAP to relational tables because the LDAP model is hierarchical. It is well known that hierarchies are very easy to represent in the hierarchical databases (like IMS) where the structure of the data and the structure of the database are the same.

Unfortunately, relational databases provide inefficient support for such data. It does not directly map hierarchical data into tables because tables are based on sets rather than on graphs. Different vendors provide different mechanisms for the tree structure. For example, DB2 provides the WITH clauses in the select statement to provide subtree traversal with arbitrary depth. Oracle has CONNECT BY PRIOR and START WITH clauses in the SELECT statement to provide partial support for reachability and path enumeration. But all mechanisms will end up with recursive queries to handle hierarchical structures. Through experiments, we discovered that recursive queries do not scale up well for large numbers of records in the table.

However, we discovered that with simple relations, we were able to support LDAP search (base, one level and subtree) with decent performance.

In addition, we also have an LDAP entry table which holds the information about an LDAP entry. This table is used for obtaining the EID of the entry and supporting LDAP_SCOPE_ONELEVEL and LDAP_SCOPE_BASE search scope. Entries are stored using a simple text format of the form "attribute: value" as in the U.M. reference implementation. Non-ASCII values or values that are too long to fit on a reasonable sized line are represented using a base 64 encoding. Giving an ID, the corresponding entry can be returned with a single SELECT statement.

4.1 Mapping LDAP Hierarchy through Relations



As illustrated in the following figure, the LDAP naming hierarchy includes a number of entries, with each entry represented by a unique entry identifier (EID). Thus, for example, the root node has an EID = 1. Root has two children, entry GB ("Great Britain") having an EID =2, and entry US ("United States") having and EID =3. Child node US itself has two children, O=IBM (with EID=4) and O=Netscape (with EID = 5). The remainder of the naming directory includes several additional entries at further sublevels.

A particular entry thus may be a "parent" of one of more child entries. An entry is considered a "parent" if it is located in a next higher level in the hierarchy. Likewise, a particular entry may be an ancestor of one or more descendant entries across may different levels of the hierarchy. A parent-child pair will also present an ancestor-descendant pair.

We created two relations to model the hierarchy in LDAP: parent-child relation (parent table) and ancestor-descendant relation (ancestor table). The parent table is created as follows. For each entry that is a parent of a child entry in the naming hierarchy, the unique identifier of the parent entry (PEID) is associated with the unique identifier of each entry that is a child of that parent entry. For corresponding parent table for the LDAP hierarchy in Figure x is illustrated in the following table. Thus, PEID 1 is associated with EID 2 and EID 3, PEID 3 is associated with EID 4 and EID 5, and so on. Each row of the parent table includes a PEID:EID pair.

The descendant table is created as follows. For each entry that is an ancestor of one of more descendent entries in the hierarchy, associating the unique identifier of the ancestor entry (AEID) with the unique identifier of each entry that is descendent (DEID) of that ancestor entry. The AEID field is the unique identifier of an ancestor LDAP entry in the LDAP naming hierarchy. The DEID field is the unique identifier of the descendent LDAP entry. Thus, in the naming hierarchy illustrated in Figure x, AEID 1 has DEIDs 2-10, because each of the entries 2-10 are also descendants of the root node. AEID 3 has DEIDs 4-10, AEID 4 has DEIDs 6-10, and so on. Each row in the descendant table thus includes AEID:DEID pair.

For LDAP search operation, the criteria of the search is specified in a search filter. The search filter typically a Boolean expression that consists of attribute name, attribute value and Boolean operations like AND, OR and NOT. User can use the filter to perform complex search operations. The filter syntax is defined in RFC 1960. In addition to the search filter, users can also specify where in the directory tree structure the search is to start. The starting point is called the base DN. The search can be applied to a single entry (a base level search), an entry's children (a one level search), or an entire subtree (a subtree search). Thus, the "scope" supported by LDAP search are: base, one level and subtree. The parent and ancestor table is used to facilitate one level and subtree searches without recursive queries. In both cases, the search begins by going into the database and using the LDAP filter criteria to retrieve a list of entries matching the filter criteria. If the search is a one level search, the parent table is then used to filter out EIDs that are outside the search scope (based on the starting point or base DN). Likewise, if the search is a subtree search, the descendant table is then used to filter out EIDs that are outside the search scope (again, based on the base DN). However, all the steps mentioned above are performed in a single SQL query. The followings are some examples of the SQL query skeleton that we used during the one level and subtree search. In these examples, <data fields> represent the SQL column name of the relations defined in the LDAP/DB2 schema, which will be described in more detail in the next section. <table list> and <where-expression> are the two null terminated strings returned by the filter translator. <root dn id> is the unique identifier of the root dn.

One-Level Search:

```
    SELECT <data fields>
         from ldap_entry as entry
          where entry.EID in (
          select distinct ldap_entry.EID
           from ldap_entry, <table list>
           ldap_entry as pchild, <list of
       tables>
           where ldap_entry.EID=pchild.EID
            AND pchild.PIED=<root dn id> <sql
       where expressions>)
```

In the one level search query, the parent table information is contained in the ldap_entry table. Since ldap_entry table also contains the entry information, we use the SQL as operator to give an alias pchild to represent the parent and child relation. Then in the where clause, "ldap_entry.EID=pchild.EID" is used to filter out entries which is not in the one level search scope.

Sub-Tree Search:

```
SELECT <data fields>
   from ldap_entry as entry
   where entry.EID in (
       select distinct ldap_entry.EID
       from ldap_entry, ldap_desc,
       <tablelist>
   where
       (LDAP_ENTRY.EID=ldap_desc.DEID AND
       ldap_desc.AEID=<root dn id>)
       ldap_entry as pchild, <table list>
   where ldap_entry.EID=ldap_desc.EID
         AND ldap_desc.AEID=%d
             <where expressions>)
```

In the sub-tree search query, the ancestor information is stored in the ldap_desc table. The inner where statement "ldap_entry.EID=ldap_desc.EID" is used to filter out entries which is not the subtree search scope.

4.2 Database Schema

The following is the detail explanation of the tables that we defined:

Entry table

This table holds the information about a LDAP entry. This table is used for obtaining the EID of the entry and supporting LDAP_SCOPE_ONELEVEL and LDAP_SCOPE_BASE search scope. The parent and child table is included in the Entry table since the all the other attributes are dependent on EID.

EID - The unique identifier of the LDAP entry. This field is indexed.

PEID - The unique identifier of the parent LDAP entry in the naming hierarchy. For example, the LDAP entry with the name "ou=Information Division, ou=People, o=University of Michigan, c=US' is the parent of "cn=Barbara Jensen, ou=Information Division, ou=People, o=University of Michigan, c=US".

DN - The distinguished name of the entry.

DN_TRUC - Truncate DN to 250 characters so that we can build indexes on this field.

EntryData - Entries are stored using a simple text format of the form "attribute: value" as in the U.M. reference implementation. Non-ASCII values or values that are too long to fit on a reasonable sized line are represented using a base 64 encoding. Giving an ID, the corresponding entry can be returned with a single SELECT statement.

Creator - The DN of the entry creator.

Modifier - The DN of the entry modifier.

modify_timestamp - Record the time when the entry was last modified.

create_timestamp - Record the time when the entry was created.

Attribute table:

One table per searchable attribute. Each LDAP entry is assigned an unique identifier (EID) by the backing store. The columns for this table are:

EID - The unique identifier of the LDAP entry.
Attribute value - Normalized attribute values.

Truncated attribute value. - If the length of the column is longer than 250 bytes, a truncated column is created for indexing. In DB2, the maximum length for a indexed column is 255 bytes. The SQL type of the attribute depends on the LDAP data type. Indexes can be created for attributes whose size are less than 255 bytes

Descendant table

The purpose of this table is to support the subtree search feature of LDAP. For each LDAP entry with an unique ID (AEID), this table contains the descendant entries unique identifiers (DEID). The columns in this table are:

AEID - The unique identifier of the ancestor LDAP entry. This entry is indexed.

DEID - The unique identifier of the descend LDAP entry. This entry is indexed.

4.3 LDAP filter to SQL Translation

This section discusses how our system translates LDAP filters [8] to various types of SQL queries. We implemented a filter translator to generate the equivalent SQL expression corresponding to an LDAP filter that can be used in the WHERE clause of an SQL SELECT statement. For all queries, the general approach is to obtain the entry EIDs which match the search criteria based on the filter from the attribute table. Then the parent or ancestor tables are used to check whether the EIDs are located in the subtree under the base dn. After getting the entry EIDs which satisfy the filter and search scope criteria, the entry data are retrieved from the LDAP entry table. However, all the operations mentioned above are performed in a single SQL query. We discovered that combining sub-queries into a single query is much more efficient than performing sub-queries independently. The combined SQL query not only saves context switching cost, but also provides SQL query optimizer with more information to come up with an optimum access plan.

LDAP filters consist of six basic search filters with the format <attribute> <operator> <value>. Complex search filters can be generated by combining basic filters with Boolean operators AND (&), OR (|), and NOT (!).

The SQL SELECT statements used by LDAP/DB2 search routines are in the following format:

Base Level Search:
```
SELECT entry.EntryData, <operational
attributes>
        from ldap_entry as entry
        where entry.EID in (
          select distinct ldap_entry.EID
          from <table list>
          where (ldap_entry.EID=<root dn
id> ) <sql where expressions>)
```

One Level Search:
```
 SELECT entry.EntryData, <operational
attributes>
       from ldap_entry as entry
        where entry.EID in (
           select distinct ldap_entry.EID
           from ldap_entry, <table list>
            ldap_entry as pchild, <list of
tables>
            where ldap_entry.EID=pchild.EID
            AND pchild.PIED=<root dn id>
<sql where expressions>)
```

Subtree Search
```
 SELECT entry.EntryData, <operational
attributes>
       from ldap_entry as entry
        where entry.EID in (
           select distinct ldap_entry.EID
           from ldap_entry, ldap_desc,
<table list>
           where
           (LDAP_ENTRY.EID=ldap_desc.DEID
AND ldap_desc.AEID=<root dn id>)
            ldap_entry as pchild, <table
list>
            where
ldap_entry.EID=ldap_desc.EID
            AND ldap_desc.AEID=%d <where
expressions>)
```

Based on the filter received, our SQL translator will generate <table list> (a list of attribute tables) and <where-expression> (the SQL expression). <root dn id> is the unique identifier of the root dn. The where clause should only be generated if <where-expression> is not the empty string and no errors where detected in the parsing the LDAP filter.

The translation rules for basic filters and Boolean filters are presented in the following sections. In the translation rules, the tablename is the SQL table for the specified attribute and columnname is the column name containing the attribute values.

4.3.1 Equality

The equality search operator locates entries with attributes exactly equal to the given value. The translation rule is the following:

LDAP filter:
(<attr> = <value>)
SQL expression:
(SELECT EID FROM: tablename WHERE columnname = 'value')

For example, the filter (sn=Jensen) is to find surnames exactly equal to Jensen. The corresponding SQL sub-query generated for this filter is (SELECT EID FROM sn WHERE sn = 'jensen').

4.3.2 Ranges

For attributes supporting ordering, LDAP filter provides inequality operators like "greater than or equal" and "less than or equal". The translation rules are the following:

LDAP filter:

```
    (<attr> >= <value>)
    (<attr> <= <value>)
```

SQL expression:

```
     (SELECT EID FROM tablename WHERE
columnname >= 'value')
     (SELECT EID FROM tablename WHERE
columnname <= 'value')
```

For example, the LDAP filter (sn >= Jesen) locates entries with surnames lexicographically greater or equal to Jesen. The corresponding SQL sub-query generated for this filter is (SELECT EID FROM sn WHERE sn = 'jensen').

4.3.3 Substring

LDAP supports arbitrary substring matching for text attributes. The user can put the wild card character (*) at the beginning of a string, the middle of the string, the end of the string, or any combination of these in the LDAP filter. The format of the substring filter is:

```
(<attr> = [<leading>]* [any]*] [<trailing>])
```

The SQL operator LIKE is used for substring matching. The SQL like operator has the following syntax:

```
column LIKE PATTERN
```

PATTERN combines string constants with wild-card characters. SQL recognizes two wild-card characters:

- %-Match zero or more characters
- _-Match any one character

We use the SQL wild character "%" for the LDAP wild card character. The LDAP substring filter "(attribute=value-with-stars)" is translated into "(SELECT EID FROM tablename WHERE columnname LIKE 'value with percents')"

For example, the LDAP filter (sn=*jensen*) is to locate surnames containing the string "jensen". The following SQL query is generated:

```
(SELECT EID FROM sn WHERE sn LIKE '%jensen%')
```

4.3.4 Approximate

The approximate search operator locates entries with attributes which sound like the given attribute value. The format of the approximate search filter is (<attr>~=<value>).

The DB2 SOUNDEX library function is used for approximate search. The SOUNDEX function returns a 4 character string that is either a CHAR or VARCHAR. The SOUNDEX function is useful for finding strings for which the sound is known but the precise spelling is not. It makes assumptions about the way that letters and combinations of letters sound that can help to search out words with similar sounds.

The LDAP search filter "(attribute~=value)" is translated to:

```
  (SELECT EID FROM SSHI.SN WHERE
SOUNDEX(columnname)
  = SOUNDEX('value')))
```

For example, the LDAP filter (sn~=jensen) locates entries with surnames that sound like 'jensen'. The following is the SQL query generated:

```
  (SELECT EID FROM SSHI.SN WHERE
SOUNDEX(sn)
  = SOUNDEX('jesen')))
```

The basic LDAP filter can be combined to form more complicated filters using the Boolean operators and a prefix notation. The '&' operator represents AND, the '|' operator represents OR and the '!' operator represent NOT.

4.3.5 Others

The attribute values specified in the LDAP filters cannot contain UN-escaped left or right parenthesis characters. The following escape combinations (backslash followed by any character) are translated as indicated.

```
 \) will translate to ) in the SQL value
 \( will translate to ( in the SQL value
 \* will translate to * in the SQL value
 \\ will translate to \ in the SQL value
 \c will translate to \c in the SQL value where c is any
other
 characters other than ) or ( or * or \
```

Any single quote characters found in the attribute value will be translated to two single quote characters since the SQL value is enclosed in single quote characters.

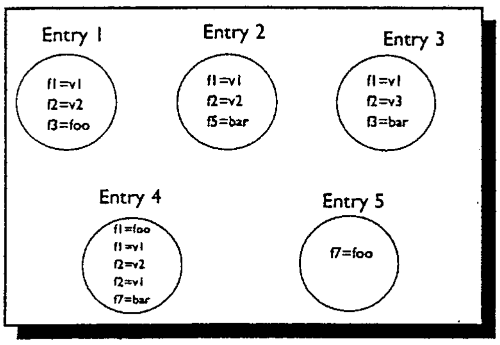
4.3.6 Complex Queries

Based on the basic translation rules mentioned above, our biggest challenges are to provide a algorithm which can do the following:

- Combine the basic expressions to form a single SQL query which will retrieve the target entries which exactly match the search criteria.

- Deal with complicated LDAP queries with infinite logical depth.

- Deal with ALL logical operators efficiently.



An intuitive solution is based on joining the attribute tables and apply the basic expressions to the attributes in the joined table. The LDAP AND (&) and OR (|) operator, in this case, can be translated into SQL AND and OR directly. In addition to the combined SQL expression, we need to include the JOIN condition based on EID.

The following is an example:

LDAP filter:
```
 ldap filter (|(f1='v1')(f2='v2'))
```

SQL Query:
```
      SELECT EntryData
      FROM ldapentry, f1,f2
      WHERE (f1.f1='f1value') OR
(f2.f2='f2value')
      AND (ldapentry.EID=f1.UID)
      AND (ldapentry.EID=f2.UID)
      AND (ldapentry.EID IN
                SELECT DEID from ldapdesc
                WHERE PEID=<UID>)
```

However, it is difficult to generalize this solution to handle the NOT (|) operator. The LDAP NOT operator is basically used to locate entries which do not match the search criteria. A
naive solution is to directly translate the LDAP NOT operator into a SQL NOT operator. The following is an example:

LDAP filter:
```
 ldap filter (!(f1='v1'))
```

SQL Query:
```
      SELECT EntryData
      FROM ldapentry, f1,f2
      WHERE NOT (f1.f1='v1')
                   AND
(ldapentry.EID=f1.UID)
                   AND (ldapentry.EID IN
                             SELECT DEID
from ldapdesc
                   WHERE PEID=<UID>)
```

The SQL query above does not yield the correct answer. The following picture illustrates the problem. There are five entries in this sample database (EID from 1 through 5) where the values of attribute f1 for entry 1,2,3 and 4 are v1. However, f1 of entry 4 is a multi-value attribute which has value v1 and foo. With the SQL statement above, entry 4 is the answer. However, the correct
answer should be entry 5. We discovered the following problems with the table join approach:

- If an entry does not contain the target attributes (for example entry 5), this entry should not be selected by the SQL statement above.

- Since an attribute can have multiple values in LDAP, the table join query will select the entry in which one of the value meet the criteria (for example, entry 4). But this entry should not be selected based on the LDAP filter.

One solution to the problems above is to retrieve all the entries from the database and filter out the candidate entry, like Netscape server. However, with a LDAP directory with large number of
entries, it takes forever to get back the results.

Another problem that we found with the intuitive solution mentioned above is that the OR operation does not perform well even for a small database with thousands of entries. Because of it is using JOIN to combine the attribute tables and ldap_entry tables. DB/2 will take a cross product all the rows in the attribute tables and ldapentry tables for the OR operation. Even though most of the rows in the cross product are irrelevant, but DB2 SQL engine dutifully reports all these rows, probably a *lot* more than are needed for the subquery evaluation.

Our solution is based on the concept of EID sets. First, generate SQL subquery for each LDAP operator based on the basic translation rules. The SQL subquery will generate a set of entry EIDs which match the LDAP basic operation. If the LDAP logical operator is OR(|), use UNION to union the sets generated from the subquery. If the LDAP operator is AND (&), use INTERCEPT to intercept the sets generated from the subquery. We experimented two different ways to put together the SQL query based on the EID set concept.

The followings are the SQL queries that our system can generate for LDAP filter (|(f1='v1')(f2='v2')):

Alternative 1:

```
SELECT entry.EntryData
 FROM LDAP_ENTRY as entry
 WHERE entry.EID in
 (
  SELECT distinct LDAP_ENTRY.EID
  FROM ldap_entry, ldap_desc, f1
  WHERE
       (ldap_entry.EID=ldap_desc.DEID AND
        ldap_desc_AID=<id>) AND
        ldap_entry.eid=f1.eid AND
        f1='v1')
        UNION
        SELECT distinct ldap_entry.EID
        FROM ldap_entry, ldap_desc, f2
        WHERE(ldap_entry.EID=ldap+desc.DEID
             AND ldap_desc.AEID=<id>)
             AND ldap_entry.EID=f2.eid
             AND f2='v2'))
```

Alternative 2:

```
SELECT entry.EntryData
FROM LDAP_ENTRY as entry WHERE entry.EID in
( SELECT distinct LDAP_ENTRY.EID FROM
LDAP_ENTRY,ldap_desc
WHERE
(LDAP_ENTRY.EID=ldap_desc.DEID AND
ldap_desc.AEID=<id>)
AND  LDAP_ENTRY.EID
IN ((SELECT EID FROM f1 WHERE f1 = 'v1')
UNION (SELECT EID FROM SN WHERE SN ='v2'
)))
```

Both SQL statements mentioned above generates the correct results. The first query is basically to perform the JOIN operation with the ldap descendant table within each subquery. The second
query is to perform the JOIN with the ldap descendant table outside the subquery. Through extensive measurement, e choose to use alternative 2 based on the performance results. In addition to correct results, the OR operation perform reasonably well with both alternative 1 and 2 since relevant entries will be filtered out in the subquery and target entries will be reported back to the main query.

With the set based approach, the NOT operation can be performed y excluding entries through negating the IN operation before he subquery. The following example illustrates the operation:

Filter String:
```
(!(f1='v1'))
```

SQL Statement:

```
SELECT entry.EntryData,
FROM LDAP_ENTRY as entry
WHERE entry.EID in
( SELECT distinct LDAP_ENTRY.EID FROM
LDAP_ENTRY,ldap_desc
WHERE (LDAP_ENTRY.EID=ldap_desc.DEID AND
ldap_desc.AEID=<id>)
AND  LDAP_ENTRY.EID NOT IN ((SELECT EID
FROM f1 where f1='v1')))
```

With the basic translation rules and the EID sets approach, we implemented a recursive algorithm which can deal with complicated queries with infinite logical operators. The following is an example of a SQL statement generated for complex query with AND, OR and NOT operator.

Filter String:

```
(&(|(objectclass=PERSON)(objectclass=GROUP)
)(sn=SMITH)(!(member=*)))
```

SQL Statement:

```
SELECT entry.EntryData,
     FROM LDAP_ENTRY as entry WHERE
entry.EID in
     ( SELECT distinct LDAP_ENTRY.EID FROM
LDAP_ENTRY,ldap_desc
     WHERE (LDAP_ENTRY.EID=ldap_desc.DEID
AND ldap_desc.AEID=?) AND
     LDAP_ENTRY.EID
     IN (((SELECT EID FROM OBJECTCLASS
WHERE OBJECTCLASS = PERSON)
     UNION (SELECT EID FROM OBJECTCLASS
WHERE OBJECTCLASS = GROUP))
     INTERSECT (SELECT EID FROM SN WHERE SN
= SMITH ) INTERSECT
     (SELECT EID FROM LDAP_ENTRY WHERE EID
NOT IN
     (SELECT EID FROM MEMBER))))
```

8. Performance

In IBM performance lab, we have measured the response time with sample databases created form white page information of IBM exployees. The size of the database ranges from a 100k entries database up to 16 Million entries database. We observed that our solution performs reasonable well compared with the industry leading LDAP server. Especially with databases with entries more than one million entries, our server outperforms the competitor. The peroformance results proved that if designed appropriately, relational tables can be used to implment a hierarchical directory structure like LDAP. An offical performance results will be published as a white paper by IBM later this year.

9. Conclusions

We have presented an efficient design and implementation to use relational database as a respository of LDAP directory information. The relational databases provide hard-won advantages such as scaleability, transaction integrity, backup and recovery, stability and a powerful query processing engine. However, Mapping the LDAP model to relational tables is not a trivial task. First of all, it appears that the LDAP model is hierarchical. It is known that hierarchies are very easy to represent in the hierarchical databases (like IMS) , where the structure of the data and the structure of the database are the same. Unfortunately, it is a general opinion that relational databases do not provide adequate support for such data. Second, LDAP allows both single and multi-valued attributes. But relational database does not deal with multi-valued attributes well. This paper presents a practical and efficient solution for the problems mentioned above. The design and implemenation mentioned in this paper is used in IBM E-Directory LDAP server. Experiments conducted in our performance lab showed that our implementation is competitive to other directory products in the industry.

Acknowledgments

Rod Mancisdor has inspired many of the ideas that we've presented in this paper. Larry Fichtner, Chin-Long Shu, Mark McConaughy, and Trung Tran also contributed greatly to the design and implementation.

References

[1] The SLAPD and SLURPD Administrator's Guide, University of Michigan, Release 3.3, April 30, 1996

[2] The Lightweight Directory Access Protocol: X.500 Lite, Timothy A. Howes, July 27, 1995, CITI Technical Report 95-8

[3] Lightweight Directory Access Protocol, W. Yeong, T. Howes and S. Kille, March 1995, RFC 1777

[4] The LDAP Application Program Interface, T. Howes and M. Smith, August 1995, RFC 1823

[5] LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol, T. Howes and M. Smith, Macmillan Technical Publishing, 1997, ISBN 1-57870-000-0

[6] A String Representation of Distinguished Names, S. Kille, March 1995, RFC 1779

[7] The String Representation of Standard Attribute Syntaxes, T. Howes, S. Kille, W. Yeong and C. Robbins, March 1995, RFC 1778

[8] A String Representation of LDAP Filters, T. Howes. June 1996, RFC 1960

[9] A Scaleable, Deployable Directory Service Framework for the Internet, T. Howes and Mark C. Smith, April 1995, CITI Technical Report 95-7

[10].Database 2, Application Programming Guide for common servers, IBM, S20H-4643-01

[11]An Efficient Relational Implementation of Recursive Relationships using path signatures, J. Teuhola, The 10th International Conference on Data Engineering, February, 1994, Houston, Texas

[12]Direct Algorithms for Computing the Transitive Closure of Database Relation, R. Agrawal and H.V. Jagadish, Proc. of 13th VLDB Conference, Brighton, England, pp. 255-266, 1987

[13]An Amateurs Introduction to Recursive Query Processing Strategies, Proc. ACM SIGMOD Confg, Washington D.C., pp16-52, 1986

[14]A Method for Hierarchy Processing in Relational Databases, P. Ciaccia, D. Maio and P. Tiberjo, Inf. Systems, Vol. 14, No 3, pp 93-105, 1989

[15]Schema and tuple trees, an intuitive structure for representing relational data, E.H. Herrin and R.A. Finkel, Computing Systems: The journal of the USENIX association, pp 93-118, 1996

[16]United States Patent, No 5467471, "Maintaining Databases by Means of Hierarchical Genealogical Table", Nov. 14, 1995