An X.500 and LDAP Database: Design and Implementation

Timothy A Howes <tim@umich.edu>

Abstract

This paper describes the design and implementation of xldbm*, an X.500 and stand-alone LDAP backend database. The* xldbm *database supports efficient execution of all queries, data modifications, and search pruning using centroids, all using simple underlying technology freely available on the Internet. Our approach to resolving various kinds of queries is described, along with a performance evaluation and comparison to other popular database packages.*

1 Introduction

X.500 [1] and LDAP [2,3] define similar directory service information and query models. The information model is centered around *entries*, which are composed of *attributes*. The entries are organized into a tree structure, usually corresponding to a geographical and organizational distribution. The query models allow searching of portions of the tree based on filter criteria involving attributes (e.g., entries with a surname of “Jensen”), and returning requested attributes from each matching entry. The model also defines operations for adding, removing, or changing entries in the directory.

The X.500/LDAP model poses some interesting challenges in database design. Searches have broad scope, spanning from one entry to the entire tree, making efficient index construction difficult. Several types of primitive searches are involved, including equality, substring, and approximate matching, and range queries. Arbitrary boolean combinations of search filters must be supported, requiring query optimization for efficient processing. Searches can span multipe servers, following aliases or not. Data can be arbitrarily distributed among servers by means of *knowledge references*. Efficient alias and knowledge handling during a search are key to good distributed performance.

We imposed an additional constraint on the underlying technology in our system. It had to be simple and freely-available Internet software. We did not want to require a commercial DBMS package, and we wanted the system to be understandable by users without much effort. Also, our goal was to make installation and administration of the system as straightforward as possible.

The database we have designed and implemented is based on any of several freely available hash or btree packages, such as GNU dbm [4], or the Berkeley db package [5]. It handles all types of X.500/LDAP searches efficiently, including full substring searches, several kinds of approximate searches, and, with an underlying ordered database (such as a btree), range queries. The database is fully disk-based and makes use of caching and threading for good performance and highly concurrent operation. It supports site-specific index configuration and performance tailoring, and centroids indexing for distributed search space pruning.

The remainder of this paper gives an overview of the X.500/LDAP information and query models driving our database design, followed by an overview of our approach to the problem. Section 4 gives details on how we handle specific types of queries. Section 5 describes our handling of aliases and knowledge references, while section 6 discusses how we use centroids to provide efficient multi-server searches. Section 7 shows how modifications affect the database structure, and Section 8 reports some performance measurements and summaries various optimizations we have implemented to improve performance. Finally, Section 9 discusses limitations of our current design, and what might be done to eliminate them.

2 Overview of X.500 and LDAP

X.500 is the OSI directory service. It defines an information model, determining the form and character of information in the directory; a namespace, allowing the information to be referred to and organized; and a functional model, determining what operations can be performed on the information.

The information model is centered around *entries*, which are composed of *attributes*. Each attribute has a *type* and one or more *values*. The type determines the attribute’s *syntax*, which defines what kind of information is allowed in the values. Entries are arranged in a tree structure and divided among servers by means of *knowledge references* in a geographical and organizational distribution. Entries are named according to their position in this hierarchy. *Alias* entries are allowed, which point to other entries, circumventing the hierarchy. Figure 1 depicts the relationship between entries, attributes, and values and shows how entries are arranged into a tree.

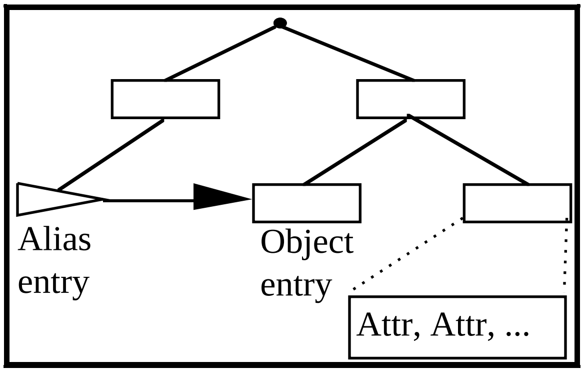

Figure 1: The X.500 model is centered around entries which arecomposed of attributes. Entries are arranged into a tree structure. Alias entries can circumvent the hierarchy.

Functionally, X.500 defines operations for searching, reading, and writing directory information. The search operation can span a single entry, an entry's children, or an entire subtree of the directory. Alias entries can be followed automatically during a search, even if they cross server boundaries. There are also operations to read a single entry or list the children of an entry. Operations are provided to add entries, delete entries, and modify existing entries.

LDAP, the lightweight directory access protocol, was originally developed as a front-end to the X.500 directory. Naturally, it assumes the same information model and namespace as X.500. LDAP is lightweight for three main reasons. First, the functional model is less complicated. The read and list operations are left out; they are emulated using search. Some of the more esoteric and less-often-used features of other operations are also not included.

Second, LDAP runs directly over TCP or other reliable transport. It avoids the overhead of the OSI session and presentation layers, making connection setup and packet handling faster and simpler.

Third, LDAP uses simple string representations for most syntaxes. While X.500 encodes data elements as highly structured ASN.1 elements, the LDAP approach encodes them as simple strings. This is a big performance win in encoding/decoding speed and complexity.

We have made minor extensions to LDAP so that it can be used as a stand-alone directory service, not just a frontend to X.500. The modifications involve the addition of referrals to the protocol, and the development of a backend database supporting the LDAP information model and query semantics. The similar X.500 and LDAP information and query models make database development for both a similar task. The same database design can be used to backend both protocols, with only minor modifications (e.g., to support the X.500 list and read operations).

3 Approach

Our approach to the X.500/LDAP database design problem is a simple one. We wanted high performance, but not at the expense of complicated management, administration, or recovery procedures. We wanted reliability, but not the complications introduced by a two-phase commit , roll-back or other guaranteed-reliable transaction protocol. We felt the system should be understandable with little effort, and easy to manage. In short, our goal was to develop a highly-functional system with good performance that was easy to administer and understand, and had reasonable reliability and recovery capabilities.

We started with the simple two-part index structure, depicted in Figure 2. First, we assign each entry a uniqne identifier by which it can be referred to efficiently. All entries in the database are maintained in a single index file, keyed by this ID. Entries are stored in this index using a simple text format of the form "*attribute: value*," as shown in Table 1. Non-ASCII values, or values that are too long to fit on a reasonably sized line are represented using a base 64 encoding, making entries in the index human readable and easy to reconstruct if lost. Given an ID, the corresponding entry can be returned quickly and efficiently, for the cost of a single hash table or btree lookup, depending on the choice of underlying technology. If other index files are corrupted or destroyed, they can be regenerated from this one file.

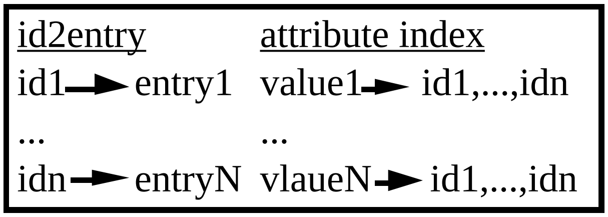

Figure 2: Xldbm index structure. The *id2entry* index stores entries in text form. The attribute indexes map values to lists of entry IDs.

Second, for each indexed attribute, we generate another file containing a list of IDs of entries containing each *value* of the attribute. (See the detailed discussion on each query type below for an explanation of what each *value* actually is.) This index is keyed by *value*, making the retrieval of a list of IDs of entries containing a given value efficient. These indexes are not text-based, nor are they meant to be read or manipulated by users (except via a low-level database administration program we provide, or indirectly through an X.500 or LDAP query). Vaues are normalized before they are added to an index (e.g., caseIgnoreString values are case-normalized, telephoneNumberSyntax values have spaces and dashes removed). The original values are retained in the id2entry index and will be returned in a search. Additional indexes of this type are constructed for Distinguished Names, aliases and knowledge references. These specialized indexes are discussed in detail later.

Table 1: Text entry representation in the *id2entry* index. The first line contains the entry ID. The second line contains the Distinguished Name. Subsequent lines contain attributes.

```
12345
dn: cn=Babs Jensen, o=Babsco, c=US
cn: Babs Jensen
cn: Barbara J Jensen
sn: Jensen
. . .
```

Given this index structure, answering a simple query is straightforward: look up the requested value in the appropriate attribute index, returning a list of entry IDs; look up those entry IDs in the *id2entry* index, returning the resulting entries to the user. For reasons discussed below, the entries read from the index files may be *candidate* entries, that is, there is no guarantee that they actualy match the query. Therefore each entry has the filter applied to it directly before it is returned as a match. This also provides an opportunity to apply access control, size and time limits, etc.

Using this simple index structure, we are able to answer virtually any X.500/LDAP query efficiently. The requirements on the underlying database are minimal: index entries are read and written; entry IDs are inserted in and deleted from index entries. Only during range queries and, optionally, some forms of approximate matching is some ordering on the indexes required, implying the btree, rather than hash file, backend.

4 Specific Query Types

This section discusses *xldbm*'s index use in detail for various types of queries. For all queries, the general approach is to consult one or more index files to generate a list of entries. Depending on the query type, these entries may have the filter applied to them directly to ensure a match. Individual search primitives are described first, followed by boolean combinations of queries.

4.1 Simple Equality

An equality search tests for entries that have a given value for a certain attribute. For example, a commonName of "Babs Jensen." Satisfying such a query using the *xldbm* index structure is straightforward: the commonName index is consulted for the list of entries corresponding to the value "Babs Jensen." Next, this list of entries is read from the *id2entry* index. The entries returned are guaranteed to match the filter.

4.2 Approximate Matching

There are several approaches to approximate matching. Phonetic algorithms such as soundex and metaphone [] are popular, and there is ongoing research into spelling or other error-correcting algorithms such as the one used by glimpse []. Currently, we support both soundex and metaphone, and have plans to support glimpse. For both phonetic algorithms we chose an approach that makes few assumptions about the structure of the data (e.g., it does not assume name data and attempt to extract a surname for matching purposes). This makes our algorithm appropriate for a wide variety of data, and reduces the risk and complexity involved in assuming a semantic structure. The disadvantage is that we are unable to take advantage of any knowledge about the type of data to improve performance.

We treat the value being matched as a sequence of words. When building the index, a phonetic code is generated for each word in the value. The code is then stored in the index, mapping to the ID of the entry containing the original value. The value given in an approximate matching query is similarly broken into words and then codes, each of which is looked up in the index.

An entry is considered to match the query if it has a value containing words corresponding to all the given codes in the proper order. If the query contains multiple words, the lists of associated IDs are intersected to produce the final list. The words must appear in the same order in the value. Since ordering information is lost in the index, the filter must be applied directly to each candidate entry to determine if it really matches the query. Table 2 shows some example approximate matching queries and values to match against (including the corresponding phonetic codes generated by the metaphone algorithm), and a brief explanation of why the value does or does not match the query.

Table 2: Example approximate matches

Query (codes)	Value (codes)	Match?
Babs Jensen (BBS JNSN)	Babs Johnson (BBS JNSN)	Yes - match
Babs Jensen (BBS JNSN)	Jensen Babs (JNSN BBS)	No - match, but wrong order
Jensen (JNSN)	Smith (SMO)	No - codes do not match
Bob Smith (BB SMO)	Bob A Smith (BB A SMO)	Yes - match
Bob A Smith (BB A SMO)	Bob Smith (BB SMO)	No - code for A is missing

There are several options for code generation and matching in the index. The simplest is to generate fixed-length codes of some maximum length. This makes generation and lookup simple. If the code length is too short, it can lead to unexpected matches because of code truncation (e.g., “Babs” matching “Babsikowjskvik” - both produce a code with BBS as the first three characters). If the code length is too long, it can lead to missed matches that should be returned (e.g., “Howe” not matching “Howes”). These two problems are in conflict.

The solution is to adopt a configurable “prefix” matching scheme in which a code is considered a match if it contains the key code as a prefix. This solves the missed match probolem, but can still lead to unexpected matches as described above. To combat this problem, we add the constraint that the two codes must differ in length by at most N characters, where N is an administrator-defined constant. Setting N to zero results in strict code matching. Setting N to a large number results in strict prefix matching (e.g., “Babs” (BBS) will match “Babsik” (BBSK) and “Babsikowjskvik” (BBSKJSKFK)). Setting N somewhere in between results in more reasonable behavior (e.g., “Babs” matches “Babsik”, but not “Babsikowjskvik”). We have found two to be a pretty good number for N.

To support this variable prefix matching requires the underlying database to support prefix retrieval of codes. With a btree or other ordered method, this is straightforward. With a hash-based scheme, it is more difficult. For small values of N and a restricted phonetic code alphabet (e.g., in the soundex scheme), it is feasible to generate all possible codes of greater length (up to N) and look them up. This method clearly does not scale well, and our implementation only implements variable prefix matching with an ordered underlying database.

4.3 Substring Matching

The substring matching problem is one of the most interesting and challenging posed by X.500/LDAP. Both models support arbitrary substring matching on text attributes. A query may specify a leading substring, trailing substring, arbitrary internal substring, or any combination of these to be matched. We set out to design a scheme that was fast, efficient and flexible. A glimpselike approach, though efficient in terms of index space used, was not fast enough and difficult to update incrementally (e.g., in response to modifications). Other approaches involving fast pattern matching on values suffer from order N performance where N is the size of the data being searched.

Our solution is to generate all substring components of a fixed length for each value and index those. Additional anchors are added to each value, marking the beginning and end of the string. When a query is presented, similar substring components are generated corresponding to it. These components are looked up in the index and the resulting lists of entry IDs are intersected to form the list of candidate entries. These candidates then have the query applied to them directly, to ensure they match the query. This last step is necessary since, as for approximate matching, the ordering of substrings is not retained in the index. Figure 3 illustrates this process for the value “Babs.”

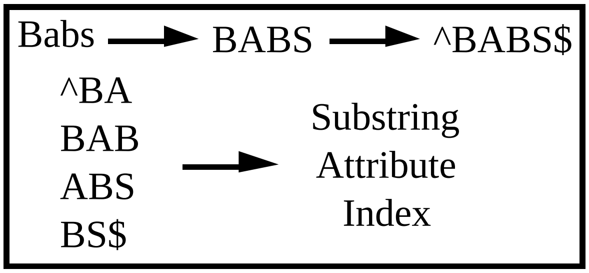

Figure 3: Substring index generation for components of length three. The value is normalized and leading and trailing anchors are added. Then all possible substrings of length three are generated and stored in the corresponding attribute index.

Our experience shows that a component length of three is optimal for databases of around 100,000 entries. The optimal length depends on the size of the data being indexed, the type of data, and a time-space trade-off in query performance versus index size. Longer components result in each substring mapping to fewer entries, but the number of distinct components increases. Shorter components reduce this number, but increases the number of entries to which each component maps. The two extreme cases provide some insight: A component length of one means that each letter is a component. For many types of data, this means that close to every entry will be listed for each component, making the list of candidate entries long. A very long component length degenerates into the equality index case, and no advantage is gained.

Note also that the component length sets a lower bound on the length of substring queries that can be supported (e.g., with a component length of three, a query for *A* cannot be answered).

In practice, this scheme works surprisingly well. For most data sets and query types, we have found they tend to contain some "power components" that help to reduce the list of candidates quickly. Pathological cases exist in which most entries are listed for each component. In such cases the "power" aspect may be contained in the component ordering, rather than the components themselves, in which case the algorithm reduces to a substring search of the entire space, as candidates are eliminated. We have found such cases to be rare in practice.

4.4 Ranges

For attributes supporting some kind of ordering, the X.500/LDAP models support inequality queries for entries containing values greater-than-or-equal-to or less-than-or-equal-to a given value. Although often the case in practice, there is no requirement that these two operators be used together to form a bounded range query. With an underlying database supporting ordered retrieval, responding to such queries is easy. With a hash-based scheme, it presents a problem we have not yet solved, except in some specific cases.

With ordered retrieval, a greater-than-or-equal-to query is answered by retrieving the given value (or "smallest" value greater than it), and then stepping through subsequent values in the ordering. The resulting lists of entry IDs are unioned together to form a single list of candidates. These candidates are guaranteed to match the filter, so there is no need to apply the filter directly to them. A less-than-or-equal-to query is handled similarly. The first item in the ordering is retrieved, followed by subsequent items until an item greater than or equal to the given key is reached. The resulting entry IDs are unioned to form the result. If the query involves range of values (i.e., greater than one vaue and less than another), obvious optimizations can be made. The efficiency of this method is proportional to the number of keys in the range.

With a hash-based scheme, ordered retrieval is not possible. In some cases, where the ordering can be approximated by a substring search, a hash-based approach can still provide results. For example, an attribute containing UTC time values has this property. A query requesting entries with a time greater than or equal to 1994 and less than or equal to 1995 produces the same results as a substring query for a time with a leading substring of "1994" (plus the simple case of a time equal to 1994). This works because UTC time has a concrete string representation that is lexicographically increasing. The situations in which this approach works are limited.

4.5 Boolean Combinations

As with any database, one of the most challenging problems is the support of arbitrary queries. If the query set is restricted and can be predicted ahead of time, design is simplified. In the case of X.500/LDAP (as with the relational model), there is no limit to the complexity of queries. The method by which these queries are built is straightforward, though, making the task easier. Boolean combinations of queries include conjunction (AND), disjunction (OR), and negation (NOT). If X and Y are queries, so are "X AND Y," "X OR Y," and "NOT X."

Conjunctive queries are easy to handle. A list of candidates is produced for each conjunction and then intersected. Note that it is not necessary to evaluate candidates before the intersection.

Disjunctive queries are similarly straightforward. A list of candidates is produced for each disjunction and then unioned. Again, candidates need not be evaluated before the union takes place.

Negation queries are more difficult. A simple approach is to produce a list of candidates matching the query and then subtract it from the list of all candidates. Unfortunately, since the original list may be only a list of *candidates* not guaranteed to match the query, blindly applying this approach can lead to lost matches. The solution is to apply the query (before negation) to the list of candidates before performing the subtraction. This produces correct results, but can be expensive.

For example if the query is for entries not containing an *objectClass* of *person* and the database contains a million entries, only one of which is not a person, the method degenerates into a linear search. In this case, it would be more efficient to step through the values of the *objectClass* attribute, building candidates from the ones matching the query. By building more knowledge into the database (e.g., how many distinct values are in an index), NOT performance can be improved.

5 Aliases and Knowledge References

Aliases and knowledge references provide similar challenges to database design. Both features create situations where a search must be continued "outside" of the original search scope, perhaps even outside the original server handling the query. Of the two, aliases are more problematic because they can point anywhere, there is no consistency requirement, and they are user-creatable.

A search that does not have the *searchAliases* flag set in X.500 or the alias flag set to *derefAlways* or *derefSearching* in LDAP is not affected by aliases. If one of these flags is set, indicating that aliases themselves should not be searched, but rather what they point to, a new phase is added to the search procedure.

The key to handling aliases is to identify those aliases that point outside the scope of the search. If an alias does not "escape" the scope of the search, the entry it points to will be searched automatically (because it is contained within the scope, not because an alias points to it - why it gets searched is immaterial, as long as it does). Once such aliases are identified, the search is continued with the entries to which they point (either the entry itself for a one-level search, or the entry and all its descendents for a subtree search). Base object searches are easy to handle by examining the entry directly, and do not require any special indexing.

To efficiently identify aliases that need searching, two new indexes are maintained, one for one-level scopes, one for subtree scopes. For each non-leaf entry, the one-level index contains an entry containing the entry IDs of alias children of the entry that do not point to other children (i.e., aliases that escape the one-level scope). Similarly, the subtree index contains entry IDs of alias descendents of the entry that do not point to other descendents. During a search, the list of candidate entries is generated as before, and then the appropriate alias-scope index is consulted to determine if there are entries outside the scope that should be searched. Figure 4 illustrates this process for a sampe tree.

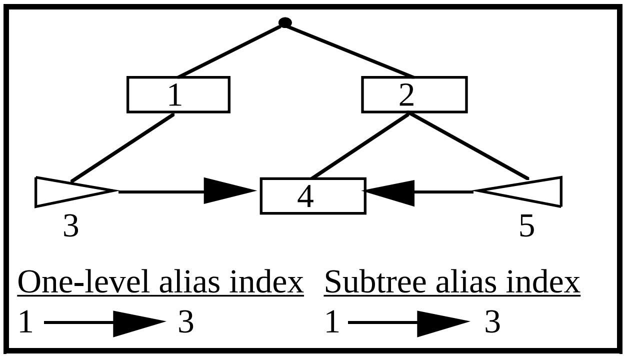

Figure 4: Alias scope index. A subtree or one-level search starting at entry 1 consultss the appropriate index and determines it needs to continue the search with the entry pointed to by entry 3.

Knowledge references are handled via a similar approach. Indexes are constructed for one-level and subtree knowledge references. Given a search scope and the entry ID of the base object, the list of knowledge references within that scope can be quickly retrieved. In X.500, these knowledge references are either used to *chain* the search or returned as continuation references to the client. In LDAP, knowledge references are returned as referrals (more on knowledge references and their relationship to centroids in Section 6). Figure 5 depicts the structure of the knowledge reference index.

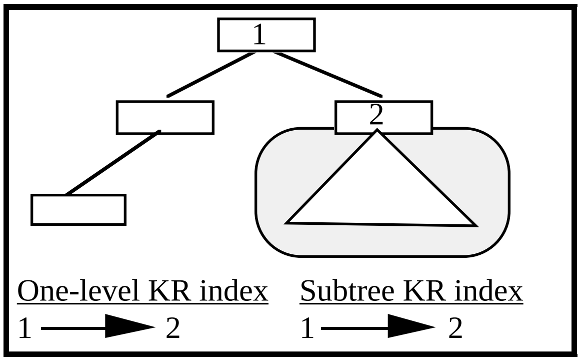

Figure 5: Knowledge reference scope index. A search beginning at entry 1 is continued at the server identified by the knowledge reference contained in entry 2.

Maintenance of the alias and knowledge reference indexes is non-trivial but straightforward. When a new alias entry is added to the tree, the one-level alias entry for its parent may need updating (if the alias does not point to a sibling entry). The subtree alias entry for each ancestor may also need updating. For knowledge references, the same is true.

This approach allows the efficient identification of alias and knowledge references at which a search must be continued. Aliases can be particularly troublesome from a performance standpoint. If many aliases escape the scope of a search, each one must be searched individually, causing a significant performance penalty. It's hard to see a general solution to this problem that guarantees good performance, and we feel this is a design deficiency in the X.500 and LDAP models.

6 Centroids

A weakness of the X.500/LDAP model is its lack of support for wide-area searches. The hierarchical scheme works well for searches whose scope can be restricted using the namespace. For searches that do not have this property, the model degenerates to a search of the entire tree, contacting every server. Clearly, this approach does not scale well.

Several solutions have been proposed, including alternate hierarchies; special "yellow pages" portions of the tree where attributes are organized to facilitate alternate searching; "alias" trees that collect pointers to information in other servers; and out-of-band distributed indexing to help prune the search space. All these schemes have their advantages, but we chose the latter approach for our system. It averts many of the maintenance and consistency probems of using aliases, and does not require the global cooperation necessary to implement an alternate namespace. It also has the advantage of presenting a consistent model to clients; they see the same tree as always, searches just happen more efficiently.

We chose to use centroids as our distributed indexing framework. Adapted from work by Salten [], and originally proposed for use on the Internet in the WHOIS++ system, centroids have the potential to provide efficient wide-area searching in the X.500/LDAP model. We have adapted centroids to this model, and included a few extensions that allow us to support the more flexible query language defined by X.500 and LDAP.

The basic centroid model involves generating the list of distinct words in a database. This list is called a *centroid* of the database. If centroids are generated for many such databases and given to another server, the server can consult the list of centroids to determine which low-level databases might hold the answer to a query. An trivial example is shown in Figure 6.

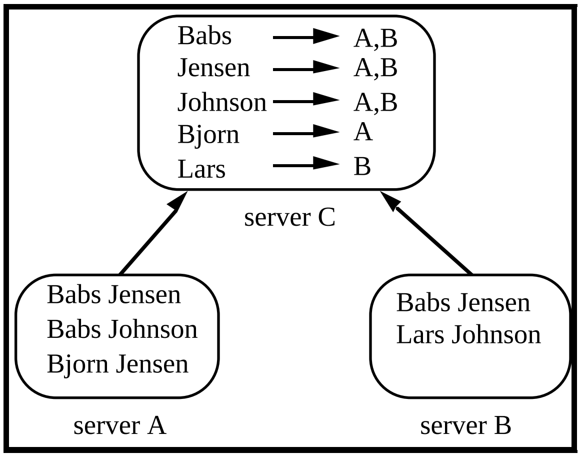

Figure 6: Centroid example. A centroid is a list of distinct words in a database. A server that collects centroids from other servers is able to determine which servers are likely to be able to answer a given query. A search involving the word "Bjorn" can be directed only to server A.

We modified the centroid model to include whole values, rather than words. Simiar modifications were made by the desighers of SOLO [], who use centroids for navigation and searching. The use of values in the centroid enables a broader range of searches to be supported (e.g., substrings), and fewer "false positives" to be returned. There is a trade-off, of course. The centroids produced are larger, not having as attractive collapsing properties as their word-oriented counterparts. Table 3 provides a comparison between the word and value approaches to centroid generation.

Tablle 3: *Word* versus *value* centroid comparison. Word-based centroids tend to be smaller, but value-based centroids allow more accurate and flexible query resolution. In this example, the word centroid is smaller by one "Babs" and one "Jensen," but gives a false positive match for a query for "Bjorn Johnson."

Values	Word centroid	Value centroid
Babs Jensen	Babs	Babs Jensen
Babs Johnson	Jensen	Babs Johnson
Babs Jensen	Johnson	Bjorn Jensen
Bjorn Jensen	Bjorn	

Incorporating centroids into the X.500/LDAP model is not difficult conceptually. A search is initiated at some point in the hierarchy. Normally, a server would search its own data and chain or refer the search to all servers holding data below it in the tree. If a server holds centroids for these servers, it can consult them and only chain or refer the search to those servers possibly able to satisfy the query. Figure 7 illustrates this process.

fix

Figure 7: Search-space pruning using centroids. A server holding centroid data for servers below it in the tree only chains or refers the search to those servers possibly able to answer the query.

In our database design, we introduce the concept of a centroid entry. Conceptually, the entry holds an entire centroid, along with access information for the server that generated the centroid. (In our implementation, this information is contained inthe name of the centroid entry.) The values in the entry are added to the indexes just like normal values. During a search, the indexes are consulted as usual, returning the ID of the centroid entry. Here the handling of centroid entries differs from regular entries. Instead of returning the entry to the client, the search is continued using the access information in the centroid entry, or the access information is returned to the client so it may continue the search.

This implementation of centroids has several advantages. First, it was very easy to implement. Once we developed the index structure described in the previous sections, it took literally less than a dozen lines of code to support centroids. Second, the pruning happens through the normal indexing consultation process. Third, the centroid generation and addition process is done using normal DAP or LDAP operations. Centroids are added using the add operation and deleted using the delete operation. Existing centroids are modified using the modify operation (e.g., in response to incremental changes to the centroid). A separate process is responsible for using these operations to generate and apply centroid changes to and from the database. This makes it easy to add new schemes later, or change the existing scheme.

7 Modifications

Both X.500 and LDAP support adding, deleting and modifying entries in the directory. They assume that read requests of the directory are far more frequent than writes, but modify performance is still an issue. In both models, every set of modifications must either succeed or fail as a group. Although we do not provide a transaction system with roll-back capability, we do minimize the time during which a serious failure can occur.

A modification is implemented as a three-step process. First, an in-memory copy of the entry is changed. If this fails, or the entry fails to satisfy the directory schema requirements after the modifications are applied, the operation is aborted. Second, the affected indexes are updated. This involves inserting and deleting entry IDs from index entries. If anything goes wrong here, the operation is aborted. Third, the *id2entry* index containing the entry itself is updated. In case of a catastrophic failure, this is the only file that is crucial. The other indexes can be reconstructed from the *id2entry* index. So, the only critical section of the modify is the update of the *id2entry* index. Since each entry is represented in the index by a contiguous sequence of characters, the update of this index can be done with a single write operation, further reducing the risk.

In practice, we find this level of reliability adequate, especially when combined with the ability to do replication, which we have implemented. A full discussion of our replication strategy is beyond the scope of this paper, but changes are logged to a file which is then read by a separate replication process, responsible for distributed changes to any number of replicas.

8 Performance Enhancements

As described, the system works surprisingly well, especially considering its simplicity. We have made several enhancements that have significantly increased performance, especially on modifies. First, our original design called for a separate index file for each attribute and index type (i.e., for each attribute files for the equality index, substring index and approximate index are created). To reduce the number of indexes open at one time, we have combined these three files into one. Doing so required the addition of a prefix scheme for storing the value keys in order to avoid conflicts. Equality keys are prefixed by "=", substring keys are prefixed by "*", and approximate keys are prefixed by "~". This reduces by a factor of three the number of open indexes necessary, with little impact on performance.

Second, our original design called for index entries as single blocks of entry IDs. In practice, these blocks become quite large. For example, in our database there are close to 100,000 people entries, causing the single *objectClass* index entry for the value *person* to be about 400,000 bytes (100,000 entries * 4 bytes per entry ID). If an entry is added, this index entry increases by one entry ID, causing the entire 400K block to be rewritten. This caused a severe performance penalty. To combat the problem, we introduced a scheme in which index entries are broken up into file system-sized block fragments. Small blocks are stored as before. Large blocks are accessed with a level of indirection through a "header" block which points to the fragments. A slight penalty is paid when the block is read (several small reads take longer than than one large one), but the performance increase on a modify is substantial. Generally, only one fragment needs updating.

Third, we noticed that some index entries, particularly in the substring index, were so large that they contributed little to reducing the number of candidates. Worse, such blocks contribute to poor performance, both on reads and writes. To reduce the effects of such large blocks, we introduced the concept of an *allIDs* block. The *allIDs* block is a simpe stand-in for a block containing all entry IDs in the database. It contains a single flag-valued entry ID, making it very small. The *allIDs* block acts like the universal set in index operations. The block size above which a block is replaced by an *allIDs* block is configurable. The optimum size depends on the database size and content. In practice, we have found a size of between 5,000 and 10,000 entry IDs to work well for our 100K-sized environment. Pathological cases exist in which this approach leads to a linear search of the database, but we have found these cases to be rare.

Finally, our biggest performance enhancement comes from extensive use of caching. The most expensive part of the search operation is the reading of candidates from the *id2entry* index, and the subsequent application of the filter. We maintain a configurable in-memory cache of entries, keyed by DN and by entry ID. Table 4 shows a comparison of some typical searches, with "cold" and "hot" entry caches. We also maintain a cache of open index files, and the underlying hash or btree database keeps a cache of its own.

Table 4: Effect of caching on performance for various queries.

Query	Cold cache	Hot cache
cn=Babs	-	-
cn=*Babs*	-	-
cn~=Babs	-	-

9 Limitations

Our scheme performs well, but it does have limitations. First, the scheme will not scale to more than a few hundred thousand to a million entries. The algorithms are sound, but the use of simple file system-based hash and btree packages incurs a performance penalty. Replacing these underlying systems with a DBMS system would undoubtedly improve performance and increase the number of entries the scheme can handle.

Second, we feel modify performance is still unacceptably poor. The limiting factor here again is the underlying database package. A file system-based approach, not optimized for modification, nor for our application in particular, simply cannot compete with a commercial DBMS.

Finally, the user-friendy text representation used to store entries in the *id2entry* index slows down the reading of entries. Especially in the X.500 case, there is significant time spent converting from the text format to an internal representation. Storing entries in BER-encoded ASN.1 (the representation used on the wire) would greatly improve this performance. It would require developing new BER-aware versions of many utility routines, including things like *strcmp* and friends.

10 Summary

This paper describes the design and implementation of *xldbm*, a backend database for X.500 and stand-alone LDAP. The system uses simple freely-available database technology and provides good performance, reliability, and recovery. *Xldbm* handles virtually all types of X.500/LDAP queries efficiently, including full substring and approximate matches. Aliases and knowledge references are handled through clever index construction. Centroids have been adapted for use as a search space-pruning device. Several performance enhancements have been implemented, making the system perform surprisingly well.

11 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. NCR-9416667.

References

[1] The Directory: Overview of Concepts, Models and Service. CCITT Recommendation X.500, 1988.

[2] W. Yeong, T. Howes, S. Kille, Lightweight Directory Access Protocol. Request For Comments (RFC) 1777, March, 1995.

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

Author Information

Tim Howes is a Senior Systems Research Programmer for the University of Michigan's Information Technology Division. He received a B.S.E. in Aerospace Engineering, a M.S.E. in Computer Engineering from U-M, and is completing a Ph.D. in Computer Science. He is currently project director and principal investigator for the NSF-sponsored WINX project, and in charge of directory service development and deployment at U-M. He is the primary architect and implementor of the U-M LDAP directory package, the DIXIE system, the GDA X.500 DSA, and a major developer of the QUIPU X.500 implementation. He is author or co-author of several papers and RFCs, including RFC 1777 and RFC 1778 defining the LDAP protocol. He is chair of the IETF Access, Searching, and Indexing of Directories working group, and an active member of the ACM and IEEE.