Nomenclator Descriptive Query Optimization for Large X.500 Environments

Joann J. Ordille `joann@cs.wisc.edu`

Barton P. Miller `bart@cs.wisc.edu`

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

Abstract

Nomenclator is an architecture for providing efficient descriptive (attribute-based) naming in a large internet environment. As a test of the basic design, we have built a Nomenclator prototype that uses X.500 as its underlying data repository. X.500 `SEARCH` queries that previously took several minutes, can, in many cases, be answered in a matter of seconds. Our system improves descriptive query performance by trimming branches of the X.500 directory tree from the search. These tree-trimming techniques are part of an active catalog that constrains the search space as needed during query processing. The active catalog provides information about the data distribution (meta-data) to constrain query processing on demand. Nomenclator caches both data (responses to queries) and meta-data (data distribution information, tree-trimming techniques, data access techniques) to speed future queries. Nomenclator relieves users of the need to understand the structure of the name space to locate objects quickly in a large, structured name environment. Nomenclator is a meta-level service that will eventually incorporate other name services in addition to X.500. Its techniques for improving performance should be generally applicable to other naming systems.

Research supported in part by an AT&T Ph.D. Scholarship, National Science Foundation grants CCR-8703373 and CCR-8815928, Office of Naval Research grant N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.

1. INTRODUCTION

Our goal is to provide descriptive (attribute-based) name queries in a large internet environment. In this paper, we describe how to provide such a facility specifically for X.500. With today's complex and growing internets, it is increasingly difficult to locate any particular resource, person, or information. The international efforts to standardize directory services, resulting in CCITT Recommendation X.500 [5], offer us the prospect of one unified system for storing naming information. Unfortunately, the unified system by itself is not enough to ease the difficulties in locating objects. Users still need to understand the structure of the name space to locate objects quickly in a large X.500 environment, because the name space is essentially hierarchical. Descriptive queries would allow users to locate objects by describing their attributes. Users are relieved of the need to understand the structure of the name space and direct searches for particular objects by navigating that name space.

X.500 has a descriptive query facility, called `SEARCH`, but its performance is limited because X.500 provides neither auxiliary data structures to constrain the search nor caches of the results of the search. For example, `SEARCH` took almost 4 minutes in our test environment to answer Query 1 in Table 1 (see Section 4). Jakobs summarizes the failure of descriptive naming in X.500:

> In how far does the present system meet the original demand for user-friendliness? Today's situation is characterized by the priority of system management aspects to user-friendliness: a global name space with distributed naming authority may not be adequately coped with by today's systems. Thus, it is the user who is left to carry the can. One possible solution might be to use descriptive names only.... However, searching in a system like this brings up problems (inter-DSA [directory system agent] communication, ambiguity of names, data management) that − at least today − cannot be solved [10].

Nomenclator meets the performance challenges of descriptive queries in a large X.500 environment.

It increases descriptive query performance by trimming branches of the X.500 directory tree from the search. It caches both data and meta-data to speed future queries. When a new descriptive query is covered by a previous query, Nomenclator answers the new query from its data cache. When a new query overlaps a previous query, Nomenclator re-uses tree-trimming techniques from its meta-data cache. These techniques result in improvements of up to 38.6 times that of X.500 in descriptive query performance when the caches are empty, and more than that for cache hits[†].

Nomenclator is a heterogeneous name service. Existing name services, like X.500, are sources of naming data for Nomenclator. By supporting heterogeneous data sources, called *data repositories*, Nomenclator relieves users of the need to understand the proliferation of name services. Nomenclator uses the relational data model to reduce complexity. A small set of global relations gives a uniform structure to existing naming data, including data from different organizations and heterogeneous sources. Users are relieved of the need to understand the structure of an underlying name space, like the X.500 naming tree. They are also relieved of the need to understand the access techniques for different types of data repositories. Unlike meta-level services that query all underlying name spaces [7], Nomenclator constrains searches to those locations where data relevant to the query is likely to exist. A relational query language including selection and projection, but not join, provides a uniform interface to the naming data in Nomenclator. One query can return an integrated view of the data from many different sources.

Two major innovations of Nomenclator improve descriptive query performance. These innovations are an active catalog that constrains query searches and extensive, multi-level caching. Components of the active catalog, called *catalog functions*, supply reusable descriptions of the data distribution. Catalog functions collect distribution information using an understanding of the syntax and semantics of the underlying name spaces. Catalog functions for X.500 use schema information and indexing techniques to trim parts of the naming tree from the search space. Some catalog functions return descriptions of the static characteristics of data distribution embedded in the X.500 schema. For example, one schema convention for X.500 places information about people in organization or locality subtrees, but not in application process subtrees [4]. Application process subtrees can immediately be trimmed from any search for information about people. Other catalog functions for X.500 return descriptions of the dynamic characteristics of data distribution by using a variety of indexing techniques on one or more attribute values. Catalog functions can be generated automatically or can be written by an organization's naming administrator.

Performance is further improved in Nomenclator by caching query responses and the descriptions of data distribution. Nomenclator extends relational database work in multiple query optimization [8] to data caching for improved descriptive query performance. New queries that are covered by the results of previous queries are answered from the data cache. Nomenclator introduces *meta-data caching* as a technique for improving descriptive query performance. In meta-data caching, information about the characteristics of the data is cached. Knowledge about search constraints, searching techniques and data access techniques is saved in the meta-data cache and re-used to improve performance.

Other components of the active catalog, called *data access functions*, encapsulate the heterogeneity of data repositories. Data access functions translate Nomenclator queries into queries that are understood by the data repositories and return data in a standard Nomenclator format. They make the query resolution algorithm independent of the access techniques for the underlying data repositories. Data access functions for X.500 map attributes in Nomenclator to attributes in X.500 objects. Sometimes these functions use inheritance to combine attributes from different levels of the X.500 tree into one tuple in a Nomenclator relation. For example, the `people` relation used in Figure 6 combines the value of the X.500 `countryName` attribute from the `country` object with the value of the X.500 `commonName` attribute from the `person` object.

The Nomenclator query resolver is a data-driven engine for locating the answer to a query. It uses the results of catalog functions to constrain the search and the results of data access functions to answer queries. The resolver retrieves both types of functions from the distributed catalog of relations. Each function is accompanied by a description of the conditions for using it, so the functions can be saved in the meta-data cache and re-used as appropriate. Catalog functions and data access functions are together called *access functions*. New name spaces and data access techniques are simple to add by creating new access functions.

The following sections describe our research in more detail. Section 2 provides an overview of the Nomenclator architecture, and Section 3 provides an overview of X.500. Section 4 describes experiments that we have run with Nomenclator on X.500, and reports the performance improvements of Nomenclator descriptive queries for X.500. Section 5 describes related work on user-friendly X.500 queries. Finally, Section 6 presents a summary.

[†]Yes, 38.6 times, 3860%.

2. THE NOMENCLATOR ARCHITECTURE

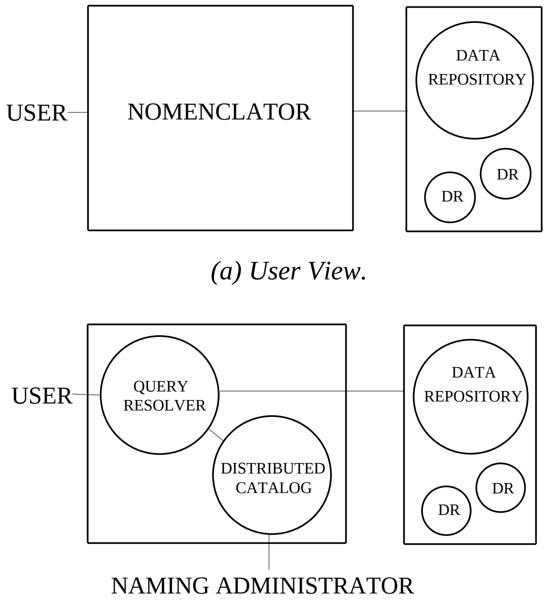
The Nomenclator architecture provides a single interface that reduces the complexity of locating objects, supports simple integration of data from a variety of sources, and improves the scaling and performance of descriptive name services. Section 2.1 provides an overview of the Nomenclator Architecture from the perspectives of users, naming administrators, and system implementors. The next three sections describe the primary mechanisms that Nomenclator uses to improve descriptive query performance. Section 2.2 describes the catalog functions that constrain the search space. Section 2.3 explains how descriptions of the data distribution, called *referrals*, are used, combined, and cached to improve performance. Section 2.4 describes advanced referral techniques that improve data access, increase cache utilization, and help users construct better queries.

2.1. Overview

Users see Nomenclator as one, uniform interface to data from a variety of sources called *data repositories* (see Figure 1a). Queries are expressed in QUEL, and specify selection and projection operations on a particular relation. Attributes common to multiple data repositories are integrated under one attribute name. Attributes that are not available in a specific data repository are given null values, and are processed using the null value techniques of GEM [23]. The responses to queries are temporary relations that integrate the data while remembering its origin. Each tuple in the temporary relation has an attribute that identifies the data repository that contributed it to the relation. This attribute, called the *source attribute*, helps users to track down and change incorrect data. If the search space for a query is too large or costly, Nomenclator returns the query with suggested revisions for improving its performance.

Naming administrators see Nomenclator as a system composed of a distributed catalog and a query resolver. These components provide access to heterogeneous data repositories (see Figure 1b). Naming administrators describe the type and distribution characteristics of their data to the distributed catalog. The distributed catalog organizes this meta-data into a form useful to the query resolver. The query resolver accepts a query from the user, and searches a constrained set of data repositories for the answer to the query. The query resolver uses information from the distributed catalog to constrain the search space and to format communication with the data repositories. Nomenclator is unique among descriptive name services, because the data providers, not the data users, supply the meta-data that guides searches.

System implementors see the internal structures and algorithms of the distributed catalog and the query resolver. The distributed catalog organizes the meta-data about the name space into definitions of relations



(a) User View.

(b) Naming Administrator View.

Figure 1.
The Nomenclator Architecture.

called *schemas*. As shown in Figure 2, each relation schema defines the name and attributes of the relation. The schema also includes a list of *referrals* that describe the *access functions* used to locate and retrieve the data in the relation. A referral contains a template and a list of references to access functions. The template is a selection predicate that describes the scope of the access functions. The access functions can be used to process queries that imply the template; they cannot be used to process queries that imply the negation of the template. For example, Query 1 in Table 1 can be processed using the first referral in Figure 2. The query resolver uses information from the schemas, in particular the referrals, to drive its query processing.

Access functions respond to the question: "Where is data answering this query?" There are two types of responses corresponding to the two types of access functions. The first response is: "Look over there." *Catalog functions* return this response; they constrain the query search by limiting the data repositories contacted to those having data relevant to the query. Catalog functions return a list of referrals to data access functions that will answer the query or to additional catalog functions to contact for more detailed information. The first referral in Figure 2 contains a catalog function depicted with the list of referrals it returns. This catalog function can locate information for any name in the country "US". The second response to "Where?" is: "Here it is!" *Data*

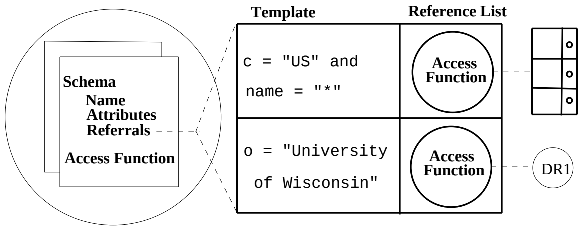


Figure 2.
The Distributed Catalog.

access functions return this response; they understand how to obtain query answers from specific data repositories. The second referral in Figure 2 contains a data access function depicted with the data repository, `DR1`, that it uses to answer queries. Referrals include addressing information for the data repositories to be contacted, and their replicas if any, thereby allowing one data access function to be used for many different combinations of data repositories.

A major goal of the query resolver is to reduce the combined cost of data location and retrieval. It accomplishes this goal in three steps. The first step checks the data cache for responses to past queries that cover the new query. If such a cached response is found, the new query is executed on the cached response to produce the desired result. If no cached responses answer the new query, the second step finds an initial set of data repositories that cover the query using referrals from the meta-data cache or, if necessary, the distributed catalog. The third step determines whether to try to reduce the set of data repositories to be searched further by contacting more catalog functions or to process the query with the current set.

2.2. Catalog Functions

Catalog functions are central to the function and performance of Nomenclator. The goal of catalog functions is to reduce the cost of locating data by constraining the data repositories to be searched. They provide an alternative to the exhaustive searches of many hierarchical systems, like `SEARCH` in X.500, and a generalization of data indices for a large internet environment. Catalog functions can be tailored to the access patterns of the whole user community or some part of it. For example, an organization might build catalog functions to speed access to portions of a relation that are important to its business needs. Catalog functions for an organization's portion of the `people` relation can use any attributes, for example `name`, that are selective within the organization. The resolver can constrain queries that cross organizational boundaries by using combinations of catalog functions from the different organizations. Catalog functions can call each other in a directed acyclic fashion to take advantage of the higher selectivity of common attribute values as the search space becomes more constrained. Catalog functions generate referrals for specific queries that fall within their scope. Generation of referrals reduces the number of referrals managed by the distributed catalog while still supplying referrals as needed to improve query performance.

Catalog functions can be implemented to execute locally, within the query resolver's address space, or remotely, via a remote procedure call to another host. The distributed catalog supplies code to the query resolver for catalog functions that are executed locally; therefore, these catalog functions can be highly replicated. Remote catalog functions may not be replicated, but they are more appropriate if the function performs a lot of work that must be amortized over a large number of users. They are also appropriate if the location of the function affects its performance as is the case when the function needs access to other centrally located data sources. Organizations may choose to supply remote catalog functions over local ones if their catalog functions use proprietary information or algorithms.

2.3. Referrals

The referrals returned by the distributed catalog and its catalog functions can often be combined algebraically to reduce the search further. If a query is a simple conjunction, like Query 4 in Table 1, multiple referrals to data access functions can be intersected to reduce the search. For example, in Figure 3,† catalog function `US_States` uses `state` to distinguish between data repositories containing information on people in the U.S. Catalog function `US_Names` uses `name` to distinguish between the data repositories. The more general catalog function `US_People` calls these two functions to constrain the search space for Query 4. It receives referral `r1` from `US_States` containing references to data repositories `DR1` through `DR8`, and referral `r2` from `US_Names` containing references to `DR7` and `DR10`. Since Query 4 is covered by these two templates, `US_People` returns referral `r3` containing the intersection of the two lists of references, i.e. a referral to `DR7`. This technique works well for simple conjunctions. Queries containing disjunctions are processed in a similar manner by converting them to disjunctive normal form [14], using catalog functions for the component conjunctions, and forming the union of the resulting reference lists.

†For the purposes of our discussion, Figure 3 uses a simplified representation of referrals: a template followed by a reference list of data repositories. For simplicity, none of the data repositories are replicated.

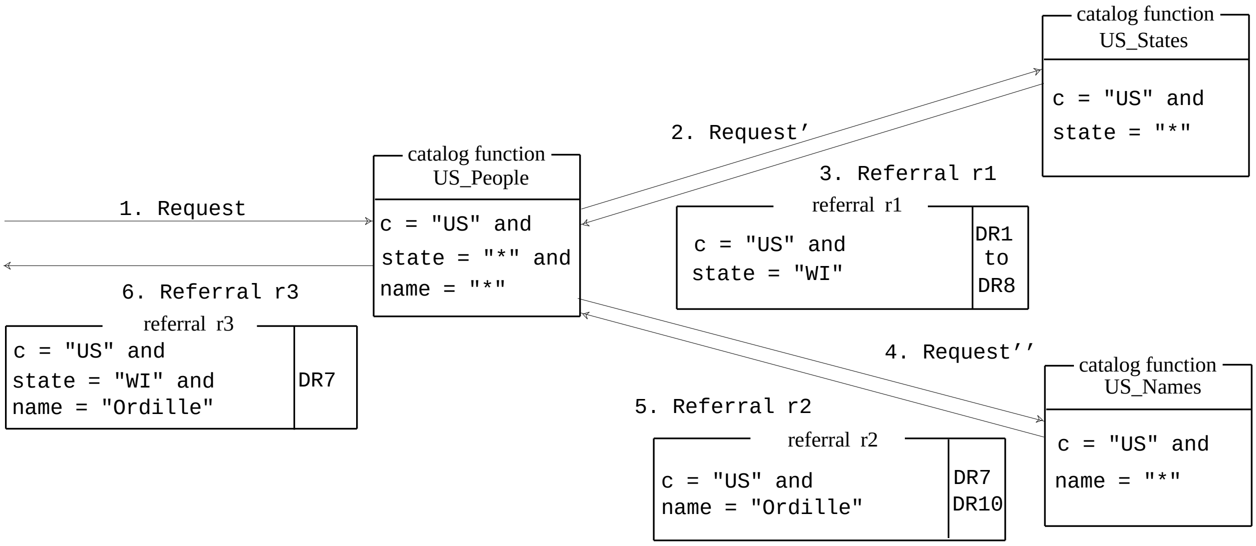


Figure 3.
An Example of Combining Referrals.
Catalog function `US_People` *intersects two referrals.*

Catalog functions must return at least one referral that covers, i.e. is logically implied by, the query. Catalog functions may also return additional referrals that might help in processing future queries. For example, in Figure 3, `US_People` can also return `r1`, `r2`, and referrals to the `US_States` and `US_Names` catalog functions. The resolver keeps referrals in its meta-data cache, and re-uses them when they help to constrain the search space for subsequent queries. For example, the resolver can re-use `r2` in processing Queries 1 and 6 in Table 1. It can re-use `r1` in processing Query 3. It can use the referrals to the catalog functions to call `US_States` directly when processing Queries 5 and 6, and to call `US_Names` directly when processing all the queries not covered by `r2`. Re-using referrals improves performance by eliminating calls to catalog functions and the distributed catalog.

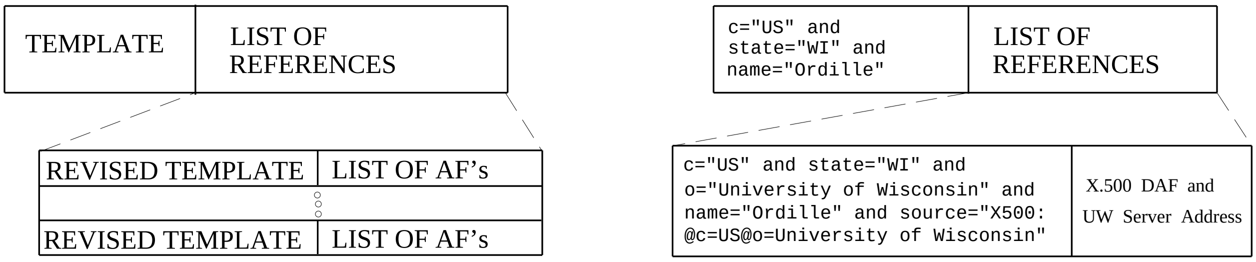
In addition to caching referrals, the resolver also maintains a data cache of query results. Like cached referrals, cached results that cover a query are re-used to answer new queries. To cover a query, a cached result must have a selection predicate that is implied by the new query. It must also contain all the projected attributes in the new query. Our techniques for using cached query results are similar to Finkelstein's [8]. We extend these techniques to the meta-data caching of referrals that describe the distribution of naming data and the conditions for using catalog functions. The caching of name server resource records (NS RR's) in the Domain Name System (DNS) [15] and prefix tables in Sprite [22] are examples of simplified versions of meta-data caching.

Because Nomenclator caches data and meta-data, a query response may not reflect the most current data available. This is not typically a problem for users, however, because most of the cached information is current and Nomenclator guarantees that caches eventually reflect the correct data and meta-data. Most cached information is current, because caches are updated at a rate that exceeds the rate of change to the information. Many naming systems, like Grapevine [2], DNS, and the QUIPU implementation of X.500 [12, 19], use this approach to replicated or cached information.

2.4. Revising Referral Templates

Catalog functions use knowledge about the structure of the name space to generate a referral. Some of this knowledge can be useful to access functions that subsequently process a query. When a referral causes an access function to be called, the referral is included in the input to the access function. A catalog function can communicate helpful knowledge by including additional, revised templates in the referral.

Referrals describe the conditions for using particular access functions to locate and retrieve data in a relation. They contain a template and a list of references to access functions, as illustrated in Figure 4a. Each *reference* in the referral contains a revised template as well as a list of access function names and their associated addressing information. The *revised template* is covered by the list of access functions in the same way that the template is covered by the list of references. By revising the template of a referral, a



(a) General Format. *(b) An Example.*

Figure 4.
Referrals.

catalog function can communicate additional restrictions on the query to access functions in the list. For example, a catalog function may determine that the selection predicate in Query 4 can be answered in X.500 from the subtree with distinguished name `@c=US@o=University of Wisconsin`. The catalog function can communicate this information by supplying the revised template shown in Figure 4b. This revised template provides additional information about the relevant `organization` and `source` of the data.

The revised template in a reference provides a context for its associated access functions. It identifies the exact conditions for calling the access functions by indicating the part of the name space within which the call is to execute. The conjunction of the query and a revised template, called a *subquery*, is the actual query processed by the access function. Subqueries help improve cache utilization and give guidance to users who need to limit queries that will be too costly. The resolver checks the caches for previous results to subqueries. Subqueries provide a list of options for users who need to constrain their queries further.

3. X.500

X.500 [5] standardizes OSI directory services for locating people and application objects. QUIPU [12], currently the most popular X.500 implementation, is used in a pilot name space including over 350 organizations in 13 countries [18]. This section provides a brief overview of the X.500 standard and some extensions to the standard used by QUIPU.

The X.500 name space is structured as a tree of objects called the *Directory Information Tree (DIT)*. Each object belongs to at least one object class. The class determines the attributes that can be present in the object. For example, in Figure 5, there is an object from the `person` class with attributes `commonName` and `surName`. Rules for which object classes appear at what levels of the DIT are not fixed by the standard. A common ordering, depicted in Figure 5, is the `root` followed by `country`, `organization`, `organizationalUnit`, and `person` objects. The standard is currently limited, because it does not describe how to store and transmit information about the structure of the DIT. QUIPU extends the standard by including an attribute in each non-leaf object, called the `treeStructure` attribute, that lists the permitted classes for children of the object [11].

Each attribute in an object is composed of a type and one or more values. At least one attribute value in each object is *distinguished*. The *distinguished values* of an object uniquely identify that object among its siblings. The path of distinguished values from the root to an object, called the *distinguished name*, uniquely identifies the object in the DIT. The distinguished name of the `organization` object in Figure 5 is `@countryName=US@organizationName=University of Wisconsin`, which is commonly abbreviated to `@c=US@o=University of Wisconsin`. X.500 supplies three commands that locate a particular object by distinguished name and return information about it. `READ` returns information about the object's attributes, `COMPARE` verifies the value for an attribute in the object, and `LIST` returns the distinguished names of the descendents of the object. All of these commands provide hierarchical access to the name space for users who can navigate the DIT.

In theory, the `SEARCH` command relieves users of the need to navigate the DIT because it searches an entire subtree looking for objects that match a selection predicate. The starting object in the search is identified by distinguished name; it could be an object as high in the tree as the `root` or `country` objects. `SEARCH` returns information about all objects in the subtree that satisfy a selection predicate called a *filter*. Unfortunately, filters only operate on attributes available in an object, not on attributes inherited from ancestor objects. Users are still forced to navigate the DIT iteratively to find paths that contain the required attributes. Moreover, searching a subtree is often

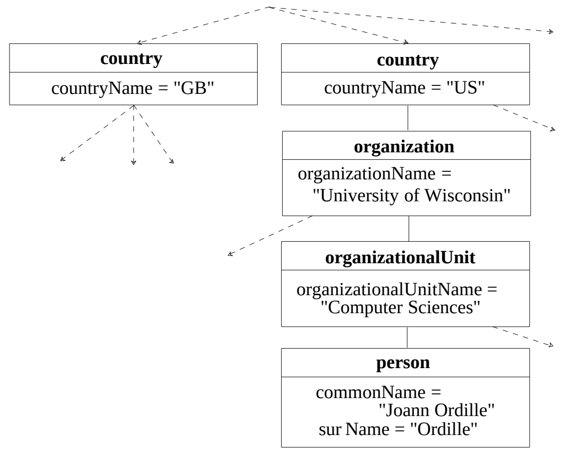


Figure 5.
Sample X.500 Directory Information Tree.

disallowed by servers higher in the DIT. Although the standard considers the country level to be "a convenient base-object for the search operation," QUIPU's default setting disallows search from this level because of the high cost of the operation. Users must navigate the DIT to search the organization subtrees of a country. We address these problems further when we describe our catalog functions and experiments (see Section 4).

The DIT is partitioned along the arcs of the tree similarly to the Domain Name System [15]. Different directory management domains are given authority for maintaining data repositories, called *Directory System Agents (DSAs)*, for different subtrees of the DIT. Each directory management domain can, in turn, delegate authority for portions of its name space. As experience with DNS has shown, caching improves data retrieval performance in this kind of highly distributed environment. Although the standard currently offers no caching support, QUIPU caches results to improve the performance of the `READ` and `LIST` commands. Nomenclator provides caching techniques for improving the performance of `SEARCH` and other descriptive queries.

4. NOMENCLATOR EXPERIMENTS IN A LARGE X.500 ENVIRONMENT

Our experiments show that Nomenclator can offer a substantial improvement to descriptive query performance in a large X.500 environment. These experiments are part of an early feasibility study of Nomenclator's approach to descriptive query optimization. They were designed to test the effectiveness of catalog functions, meta-data caching, and data caching in improving performance. In our experiments, catalog functions provide up to 38.6 times the performance of the X.500 `SEARCH` command. Data caching provides even greater improvements to the performance of `SEARCH`.

These initial experiments do not examine the cost of building and maintaining catalog functions, the cost of transferring function definitions and referrals from the distributed catalog to the query resolver, or the cost of cache management. We plan to address these issues in our continuing research. Our initial experiments are a proof of concept and are not intended as a definitive implementation; they show that constraining the search space and caching are effective ways to improve descriptive query performance.

The following sections describe our experiments in more detail. Section 4.1 describes the test environment, and Section 4.2 describes the catalog functions used in the Nomenclator prototype. Section 4.3 presents our results.

4.1. X.500 Environment

Our experiments were performed on the United States portion of the QUIPU pilot X.500 name space [19] in early 1991. At that time, the U.S. name space included 78 organizations and more than 64 DSAs. The administrators of the pilot name space reported an average of 2400 entries per U.S. organization in December, 1990 [18].

Our experiments use a prototype implementation of Nomenclator that processes queries like those in Table 1. It includes a query resolver, but no distributed catalog. The referrals that would typically be returned by the distributed catalog are preloaded into the meta-data cache, and the access functions definitions are compiled into the resolver. Our prototype allows us to measure the performance improvements that result from using catalog functions and referrals.

We also developed a prototype X.500 utility for our experiments. This utility accepts QUEL queries like those in Table 1, and returns results in the same format as the Nomenclator prototype. The response for Query 1 in Table 1 is given in Figure 6. We implemented the X.500 utility for two reasons. First, we needed to work around the restriction QUIPU places on searches starting at the country level of the DIT. The prototype keeps a list of organizations in the U.S., and it submits a `SEARCH` command to every organization subtree. If the QUEL query supplies an organization name, the utility only submits the query to that organization. Second, the utility enhances the functionality of X.500 by inheriting attributes from upper levels of the DIT. The utility can select objects for the result based on inherited as well as non-inherited attributes.

The prototypes for Nomenclator and the X.500 utility both use the same software to obtain results from an X.500 subtree. Both prototypes process

```
Name:            Joann Ordille
  Email:         joann@cs.wisc.edu
  Department:    Computer Sciences
  Organization:  University of Wisconsin
  State:         Wisconsin
  Country:       US
  Source:        X500:@c=US
                 @o=University of Wisconsin
                 @ou=Computer Sciences
                 @cn=Joann Ordille
```

Figure 6.
Result Produced for the Query 1 in Table 1.

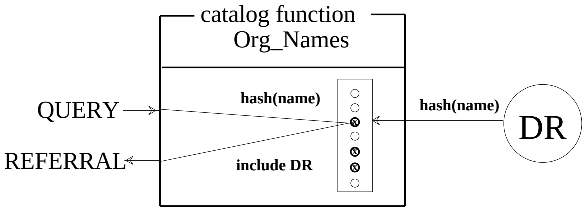


Figure 7.
A Catalog Function with a Bit Vector Filter.

queries to different parts of the name space sequentially to isolate the effects of catalog functions and caching from the effects of parallel query processing. The use of catalog functions and enhanced caching in Nomenclator are the significant differences between the prototypes.

4.2. Nomenclator Catalog Functions

The Nomenclator prototype includes `US_States` and an expanded version of `US_Names` from Figure 3. The `US_States` catalog function is implemented as an index. It maintains a table of full state names and state abbreviations. For each state, it lists the data repositories where one or more objects have a matching X.500 `stateOrProvinceName` attribute.

The expanded version of `US_Names` uses an optional `organization`, as well as `name`, to distinguish between data repositories. `US_Names` builds referrals by combining the results of calls to the `Org_Names` catalog function for each organization. Each `Org_Names` catalog function returns a referral to the data repositories in its organization if the value of `name` can be found there. If a particular organization is mentioned in a query, `US_Names` returns a referral to the `Org_Names` catalog function for that organization.

`Org_Names` hashes the values of the X.500 `surName` attribute in the data repositories for an organization. The hash values are recorded in a bit vector filter (see Figure 7). When a query is submitted to `Org_Names`, it hashes the value of the `name` attribute. If the bit for the hash value of `name` is set in the filter, the catalog function returns a referral to the data repositories in the organization. If the bit is not set, the data repositories for the organization are eliminated from the search. A bit vector filter provides a compact representation of the values of attributes in data repositories. `Org_Names` uses a filter of 20,000 bits for an organization, and a hash function on the first four letters of `surName`. The filter and the hash can be tailored to the distribution characteristics of the data in the organization. Bit vector filters have been used effectively in distributed join processing [1, 3, 6, 21]; Nomenclator is the first to apply them to distributed selection predicate optimization.

Indices and bit vector filters are general techniques that can be used by catalog functions for other attributes. It would also be interesting to build catalog functions that use more knowledge about the structure of the DIT to constrain the search space. The `treeStructure` attribute provided inadequate information for this task, because naming administrators typically took a liberal approach to the attribute. They listed all the logical possibilities for subtrees in `treeStructure`. It would be useful to have information about the kinds of objects and attributes that actually exist in a subtree, as well as the domains of values for those attributes. This information would help us to build catalog functions more effectively.

4.3. Experiment Results

Our experiments were done from a DECstation 3100 with 24MB of memory. We ran a series of queries in each experiment from the queries shown in Table 1. All of the queries are covered by the Nomenclator `US_Names` catalog function. For Queries 7-8, `US_Names` returns the `Org_Names` catalog function specific to the University of Wisconsin. Queries 3-6 are also covered by the `US_States` catalog function.

In the first set of experiments, we restarted the X.500 server (DSA) before each query to clear any caches. We also restarted our X.500 utility and Nomenclator before each of these experiments. Since we were doing experiments on the existing X.500 subtree in the U.S., we were not able to restart every X.500 server that we contacted before each experiment. In the second set of experiments, we restarted the X.500 server and ran each query twice in succession to determine the effect of caching on query performance. The X.500 utility took advantage of the DSA cache, but it did not use any cache of its own. QUIPU's DISH utility does offer an additional cache. The performance of this cache was not tested in our experiments. Nomenclator used the DSA cache, and its local data and meta-data caches.

No.	Query
1	`retrieve (people.all) where people.c = "US" and people.name = "Ordille"`
2	`retrieve (people.all) where people.c = "US" and people.name = "Miller"`
3	`retrieve (people.all) where people.c = "US" and people.state = "WI" and people.name = "Miller"`
4	`retrieve (people.all) where people.c = "US" and people.state = "WI" and people.name = "Ordille"`
5	`retrieve (people.all) where people.c = "US" and people.state = "HI" and people.name = "Miller"`
6	`retrieve (people.all) where people.c = "US" and people.state = "HI" and people.name = "Ordille"`
7	`retrieve (people.all) where people.c = "US" and people.name = "Miller" and people.o = "University of Wisconsin"`
8	`retrieve (people.all) where people.c = "US" and people.name = "Chaillou" and people.o = "University of Wisconsin"`

Table 1.
The Test Queries and Their Identifying Numbers. (queries expressed in QUEL [20])

Table 2 gives the results of our experiments. The number of items returned by each query is listed. The cold cache performance measurements are listed for each program along with the number of servers they contacted in processing the query. The cold cache results are the best times from several runs for a common baseline of servers, and include the cost of establishing a connection to the local X.500 server. We compare the X.500 and Nomenclator cold cache results in the "Improvement Factor" column. Dividing the X.500 performance by the improvement factor gives the Nomenclator performance. The X.500 warm cache results had worse performance than the cold cache results, so they are not presented here. Nomenclator's results for the warm cache are also given in Table 2. These results measure the performance of a second execution of the query immediately following its first execution. We averaged thousands of warm cache experiments to improve the granularity of these time measurements. The warm cache results do not include the cost of connecting to the DSA, because the second query is completely answered from the cache, without contacting the DSA. Finally, Table 2 includes a column that gives the improvement factor of our cached queries with respect to the X.500 cold cache results.

We find the improvement factors for a warm cache in Table 2 somewhat embarrassing. We must emphasize that these experiments were designed as a proof of concept. The caching numbers, in particular, represent a lower bound that does not include searching times for caches of substantial size. There is only one query response in the data cache when the second query is executed. We are continuing our work on cache management strategies, and plan to do more extensive experimentation of cache performance.

The improvement factors for the cold caches are more realistic. They reflect the benefits accrued in Nomenclator from trimming the search space. Note that the queries with the greatest improvement in cold cache performance are those where the number of servers contacted is most reduced. The lower improvement in Query 2 reflects the predominance of the data transfer costs in executing this query. It illustrates that the cost of contacting data repositories with no data relevant to the query becomes insignificant as the volume of data transferred rises. Our continuing research will develop techniques to aid users in reducing the volume of data transferred. We will enhance Nomenclator to identify queries with high data transfer costs and suggest subqueries with lower costs. We will also provide an iterative query that allows users to browse a large result by retrieving the answer in small increments. The iterative query will process an ordered list of subqueries that cover that original query; users will specify the order in which subqueries are processed or use a default order.

It is understandable that the DSA caching did not improve the performance of our X.500 queries, because caching in QUIPU is designed to improve the performance of the `READ` and `LIST` commands. The performance of QUIPU `SEARCH` commands should improve by incorporating our data caching techniques. As Table 2 illustrates, there is much to be gained from data caching.

We are continuing to study the effects of meta-data caching on performance. We expect referral caching to be more beneficial in an environment where catalog functions are remote to the query, or where the query resolver must be selective about which catalog functions are kept locally. Our measurements show that for the local environment of our

Query	Items	X.500 Time	X.500 Servers Contacted	Nomenclator (cold cache) Time	Nomenclator (cold cache) Servers Contacted	Nomenclator (cold cache) Improvement Factor	Nomemclator (warm cache) Time	Nomemclator (warm cache) Improvement Factor
1	1	221.4	34	13.5	1	16.4	.004	55,350
2	160	1563.8	34	1468.5	11	1.1	.256	6,109
3	2	324.0	34	15.9	1	20.4	.008	40,500
4	1	309.9	34	13.4	1	23.1	.004	77,475
5	0	351.0	34	9.1	0	38.6	.003	117,000
6	0	218.7	34	9.5	0	23.0	.003	72,900
7	2	16.1	1	15.4	1	1.1	.008	2,013
8	0	10.5	1	7.0	0	1.5	.003	3,500

Table 2.
Performance of X.500 (Cold Cache), and Nomenclator for the Test Queries.
X.500 warm cache results are not included, since (for the QUIPU implementation)
the performance was worse than for the cold cache.
(times in seconds)

tests, retrieving a referral from the cache and regenerating it are comparable in performance.

4.4. Life in the Current X.500 Environment

The current X.500 environment is still experimental and this is reflected in our measurements. We found that we could only successfully query about 35 of the 78 organization subtrees in the US. Many were unavailable for prolonged periods; others were eliminated because their administrative limits on the time and size of queries were too low to allow our tests to complete successfully.

We also found that there is much inconsistency in the use of attributes. For example, some organizations place room and phone numbers in the `commonName` of the `person` object. While most put the city followed by the state in the `locality` attribute, some put their company's division name in that attribute. The variety of interpretations of attributes made our programming task more challenging. The use of integrity constraints on the name space, like enforcing a domain of possible values for an attribute, would reduce the difficulties that result from differing interpretations of attributes.

5. RELATED WORK

Nomenclator simplifies the use of X.500 naming by improving descriptive query performance for large X.500 environments. Two other projects have similar goals to improving descriptive query performance for X.500.

Neufeld [16] suggests *registering* distinguished attribute values to improve the performance of descriptive queries. A registered value can only appear once in the subtree of its parent. For those descriptive queries that use registered attributes, search speeds are improved because we are guaranteed that registered values only exist in certain subtrees of the name space. The higher in the tree a value is registered, the more helpful it is in constraining the search and the more restrictive it is to subordinate trees. Only attributes in the distinguished name for an object speed the search, because only they can be registered.

This approach basically supports an enhanced distinguished name that allows users to supply a subset of the distinguished attributes in in an arbitrary order. No performance results are available for Neufeld's system, but when registered attributes restrict the search to one data repository, the performance of Neufeld's system should be close to our results. Nomenclator could describe and use the search constraints on registered attributes in improving the performance of descriptive queries; however, we prefer a more general approach. We can constrain a descriptive name search by using other distribution patterns, including those that characterize attributes that are not embedded in the distinguished name. Nomenclator improves performance for a wider range of queries without requiring naming administrators to constrain the attribute values for all the organizations in a subtree of the name space.

Kille [13] suggests a convention for expressing names as a list of ordered values without explicit attribute types. Heuristic search techniques resolve these names by guessing both the relevant attribute types and a reasonable subtree to search for an answer. Users are freed from knowing attribute names, but still are required to know a reasonable order to specify attributes. This is an interesting approach and can complement the design of Nomenclator. The typeless (attribute-less) interface could be built on top of Nomenclator, and Nomenclator optimization techniques can be used to improve the performance of the descriptive queries generated by the heuristic searches. Kille's approach relies on users to supply information to guide query resolution; Nomenclator uses information gathered from the name space and supplied by naming administers to guide its searches.

Profile [17] is another system designed to provide attribute-based naming although not specifically for an X.500 environment. Profile users specify preferences concerning the importance and use of attributes in the query resolution search. Some of these preferences increase performance by constraining the scope of the search while others specify ways to interpret attributes in the name space. Nomenclator differs from Profile in that the administrators, not users, specify information that guides the search.

6. SUMMARY

Nomenclator is able to provide efficient descriptive naming for widely distributed data in an X.500 environment. Nomenclator allows the owners of the data (naming administrators) to provide clues, in the form of catalog functions and referrals, to direct queries to a small number of servers. These catalog functions are simple to write, and in many cases (for commonly-used name services), will be generated automatically. In our experiments, Nomenclator used catalog functions to provide up to 38.6 times the performance of X.500 `SEARCH` commands. Nomenclator also uses extensive caching to improve performance. In our tests, data caching provided thousands of times the performance of `SEARCH` for queries covered by cached results. Meta-data caching did not improve performance in our test environment, but it will be useful in environments where catalog functions and the distributed catalog are remote from the resolver.

Nomenclator is a meta-level name service that will eventually incorporate other sources of naming data, like DNS, finger [9], and assorted relational databases. Nomenclator relieves users of the need to understand the proliferation of name services and the details of the structure of the name space. Nomenclator's techniques for improving descriptive query performance can be incorporated into existing services, like X.500. While our implementation effort is still in the early stages, the results strongly suggest that these techniques can offer a useful addition to standard services.

7. ACKNOWLEDGMENTS

We are grateful to David Presotto of Bell Laboratories for his contributions to the unity of catalog and data access functions.

8. REFERENCES

[1] E. Babb, ''Implementing a Relational Database by Means of Specialized Hardware,'' *ACM Transactions on Database Systems* **4**(1)(March 1979).

[2] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder, ''Grapevine: An Exercise in Distributed Computing,'' *Communications of the ACM* **25**(4), pp. 260-274 (April 1982).

[3] K. Bratbergsengen, ''Hashing Methods and Relational Algebra Operations,'' *Tenth International Conference on Very Large Data Bases*, Singapore, pp. 323-333 (August 1984).

[4] International Telegraph and Telephone Consultative Committee (CCITT), ''Annex B: Suggested Name Forms and DIT Structures,'' *Recommendation X.521*, pp. 295-298 (1988).

[5] International Telegraph and Telephone Consultative Committee (CCITT), ''The Directory,'' Recommendations X.500, X.501, X.509, X.511, X.518-X.521 (1988).

[6] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna, ''GAMMA - A High Performance Dataflow Database Machine,'' *Twelfth International Conference on Very Large Data Bases*, Kyoto, pp. 228-237 (August 1986).

[7] R. E. Droms, ''Access to Heterogeneous Directory Services,'' *Ninth Joint Conference of IEEE Computer and Communications Societies (INFOCOMM)*, San Francisco, pp. 1054-1061 (June 1990).

[8] S. Finkelstein, ''Common Expression Analysis in Database Applications,'' *ACM SIGMOD International Conference on Management of Data*, Orlando, FL, pp. 235-245 (June 1982).

[9] K. Harrenstien, ''Name/Finger,'' Request for Comments 742, DDN Network Information Center, SRI International, Menlo Park, CA (December 1977).

[10] K. Jakobs, ''The Directory - Evolution of a Standard,'' *IFIP TC6/TC8 Open Symposium on*

Network Information Processing Systems, Sofia, Bulgaria, pp. 281-289 (May, 1988).

[11] S. E. Kille, ‘‘The QUIPU Directory Service,’’ *Fourth International Symposium on Computer Message Systems*, Cosa Mesa, California, pp. 173-185 (September 1988).

[12] S. E. Kille, C. J. Robbins, M. Roe, and A. Turland, ‘‘QUIPU,’’ *The ISO Development Environment: User’s Manual* **5**(January, 1990).

[13] S. E. Kille, ‘‘Using the OSI Directory to Achieve User Friendly Naming,’’ Internet Draft, University College London (January, 1991).

[14] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, NJ (1981).

[15] P. V. Mockapetris, ‘‘Domain Names - Concepts and Facilities,’’ Request for Comments 1034, DDN Network Information Center, SRI International, Menlo Park, CA (November 1987).

[16] G. W. Neufeld, ‘‘Descriptive Names in X.500,’’ *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, Austin, Texas, pp. 64-71 (September 1989). Published as *Computer Communications Review* **19**(4).

[17] L. L. Peterson, ‘‘The Profile Naming Service,’’ *ACM Transactions on Computer Systems* **6**(4), pp. 341-364 (November 1988).

[18] C. J. Robbins, ‘‘The Pilot DIT,’’ Technical Report, University College London (December, 1990).

[19] M. T. Rose, ‘‘Realizing the White Pages using the OSI Directory Service,’’ Technical Report 90-05-10-1, Performance Systems International, Inc., Reston, VA (May, 1990).

[20] M. Stonebraker, P. Kreps, E. Wong, and G. Held, ‘‘The Design and Implementation of INGRES,’’ *ACM Transactions on Database Systems* **1**(3), pp. 189-222 (September 1976).

[21] P. Valduriez and G. Gardarin, ‘‘Join and Semi-join Algorithms for a Multiprocessor Database Machine,’’ *ACM Transacations on Database Systems* **9**(1), pp. 133-161 (March 1984).

[22] B. Welch and J. Ousterhout, ‘‘Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem,’’ *Sixth International IEEE Conference on Distributed Computing Systems*, Cambridge, MA, pp. 184-189 (May 1986).

[23] C. Zaniolo, ‘‘The Database Language GEM,’’ *ACM SIGMOD International Conference on Management of Data*, San Jose, CA, pp. 207-218 (May 1983). Published as *SIGMOD Record* **13**(4).