DESCRIPTIVE NAME SERVICES FOR LARGE INTERNETS

by

JOANN JANET ORDILLE

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1993

ACKNOWLEDGEMENTS

One learns many things while doing a doctoral dissertation in computer science, some technical things and some things about the world. I thank my advisor, Professor Barton Miller, for everything he taught me. I thank committee members Professor Michael Carey, Dr. David Presotto of Bell Laboratories and Professor Marvin Solomon for scrupulously reading my thesis. I thank the whole committee, including non-reading members Professor Mary Vernon and Professor Leora Weitzman of Philosophy, for an enthusiastic and thorough discussion during my defense. Special thanks to Mary for several helpful discussions of modeling issues, and to Mike and Dave for flying to Madison from opposite coasts.

I am grateful to several people who provided special support and inspiration during my graduate studies: Professor Kristin Bennett of RPI, Professor Anne Condon, Dorothy M. Corrigan, Anne Edwardson, Edith Epstein, Michael Inners, K. Dianne Maki, Professor Robert Netzer of Brown, and Dr. Ravi Sethi of Bell Laboratories. I remember and thank the late Professor Lolas Halverson for nurturing me and a whole generation of women students at the University of Wisconsin. I am also grateful to Dr. Vikram Adve of Rice, Ted Faber, Alain Kagi, Dr. Scott Leutenegger of NASA, and Mitchell Tasman for our discussions about analytical modeling, to James Elliott for our discussions about name services, and to James Elliott and Cheryl Thompson for their work on the Nomenclator software.

I thank the people of Bell Laboratories whose enthusiasm for my work and their own is a source of inspiration to me, and the people of St. Paul’s whose love and admiration is a source of encouragement. I also thank my parents, Jeanette and Joseph, my family and friends for their steadfastness, companionship and cheerfulness, and the AT&T Foundation for the support of an AT&T Bell Laboratories Ph.D. Scholarship.

ABSTRACT

This thesis addresses the challenge of locating people, resources, and other objects in the global Internet. As the Internet grows beyond a million hosts in tens of thousands of organizations, it is increasingly difficult to locate any particular object. Hierarchical name services are frustrating, because users must guess the unique names for objects or navigate the name space to find information. Descriptive (i.e. relational) name services offer the promise of simple resource location through a non-procedural query language. Users locate resources by describing resource attributes. This thesis makes the promise of descriptive name services real by providing fast query processing in large internets.

The key to speed in descriptive query processing is constraining the search space using two new techniques, called an *active catalog* and *meta-data caching*. The active catalog constrains the search space for a query by returning a list of data repositories where the answer to the query is likely to be found. Components of the catalog are distributed indices that isolate queries to parts of the network, and smart algorithms for limiting the search space by using semantic, syntactic, or structural constraints. Meta-data caching improves performance by keeping frequently used characterizations of the search space close to the user, thus reducing active catalog communication and processing costs. When searching for query responses, these techniques improve query performance by contacting only the data repositories likely to have actual responses, resulting in acceptable search times. Even a request to locate a person with a popular name in the global name space can often be answered in seconds.

The new techniques are integrated with existing data caching techniques through a single mechanism, called a referral. *Referrals* describe the conditions for using active catalog components, or re-using meta-data or data cache entries. Referrals form a DAG, called a *referral graph*, that directs query processing. The referral graph provides a uniform framework for identifying the specific system resources (e.g. cache entries or catalog components) that optimize processing for a query. It identifies search spaces that can be combined through intersection or union. It combines indices specific to organizations into a composite index for the global name space. It eliminates components with overlapping functions from query processing. The referral graph also provides a natural source of advice to users. Users who present expensive queries can be asked for specific attribute values to constrain the search for responses and lower processing cost.

Referral graphs, an active catalog, meta-data and data caching are combined in the prototype descriptive name service called Nomenclator. In one measurement study, Nomenclator consistently improved the performance of descriptive queries in X.500. Another measurement study shows how Nomenclator uses a small investment of network bandwidth and server resources to improve the response time for a wide range of query sizes. An analytical modeling study shows that Nomenclator can amortize this investment over many queries to provide an overall reduction in system load and, as a consequence, better scaling and response time. Nomenclator provides a uniform interface to various name services, such as X.500 and the Ninth Edition Unix Name Service from Bell Labs. Nomenclator is simple to extend to include new name services.

Chapter 1

INTRODUCTION

1.1. Motivation

The global Internet provides users with an information labyrinth – rich in resources, yet confusing and difficult to navigate. Many researchers are responding with a desire to integrate all files, indeed all information, into one global directory tree. Experience searching for lost files in our own local file directories tells us that hierarchical navigation is an inadequate query facility. Users must either guess the unique pathname for an object, or must navigate and list the directory tree to find the object. Descriptive (i.e. relational) name services provide an alternative that allows users to locate objects by describing their attributes. Descriptive name services offer users the hope of breaking through the labyrinth to access information quickly and directly.

Descriptive name services have, however, suffered from notoriously poor performance. The most frequent query for any descriptive name service is the global selection query, such as the query in Figure 1.1. To answer a global selection query, the name service must locate objects within a search space of tens or even hundreds of thousands of database partitions. The scale, autonomy, and availability characteristics of large internets make brute force selection techniques, exhaustive searches, and global synchronization impossible. The bottleneck for query performance is not disk access or CPU power, but communications latency and throughput. No previous descriptive name services have tackled these problems to produce a service with usable performance.

Experiences with implementations of the X.500 directory standard [22] highlight the difficulties with the performance of global selection predicates. 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 four minutes in our test environment to find people named "Ordille" in the United States, and would have taken much longer in a search of the world. Jakobs summarizes the

```
select * from People
  where
    name = "Ordille";
```

Figure 1.1. Sample descriptive query in SQL

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 [47].

This thesis provides solutions to the performance problems that plague descriptive name services, like the X.500 `SEARCH` implementations, and the user interface frustrations that plague hierarchical or graph-based name services, like Gopher [3, 28], Alex [15], Prospero [73], DNS [66], World-Wide Web [7], and X.500's hierarchical `READ` and `LIST` commands.

1.2. Summary of Results

Three techniques in this thesis improve the performance of descriptive name services in large internets. First, we introduce *active cataloging* as a mechanism to isolate queries within a subset of the data repositories that store a relation. The active catalog constrains the search space for a query, eliminating the overhead of contacting data repositories that do not contribute to the query answer. Second, we introduce *general meta-data caching* to reduce processing and communications overhead. Caching meta-data at the query site responds to the locality of user queries by retaining components of the active catalog and storing results that constrained previous queries. It eliminates the overhead of repeatedly contacting the active catalog for query constraint information. Third, we introduce *referral graphs* to integrate components of our active catalog, our meta-data caching techniques, and existing data caching techniques into one framework for fast descriptive query processing.

The active catalog structures its indexing facilities into *catalog functions* that accept a query and return a constrained search space for the query. Some catalog functions use relatively static information to constrain the search, like knowledge about the conditions used to distribute data to data repositories (called the *partitioning criteria* of a relation). Other catalog functions build indices or hash filters [4] to capture the distribution patterns in changing data, or dynamically search the network for information to speed query processing. Still others use semantic constraints like information about integrity constraints or the domains of attributes to constrain the search.

Information in the active catalog is intelligently replicated in meta-data caches to tailor query sites to the types of queries they see most frequently. Intelligent replication is a partial replication; no one site contains the entire contents of the active catalog but rather those parts that are currently most useful to it. Information on which catalog functions to use and the constrained search spaces that result from using catalog functions are cached for subsequent use. When searching for query responses, our techniques often restrict the search to the small percentage of data repositories with actual responses, resulting in performance improvements of as much as an order of magnitude.

Meta-data descriptions, called *referrals*, specify the conditions for using catalog functions and other active catalog components. Referrals form a DAG, called a *referral graph*, that directs query processing. The referral graph provides a uniform framework for identifying specific system resources that optimize processing for a query. It provides one mechanism for identifying meta-data or data cache entries that answer a query, thereby uniting our new meta-data caching techniques with existing data caching techniques. It allows us to select the right catalog function for our needs and reap the benefits of multiple catalog functions in processing one query. The referral graph is the organizing principle that directs the creation, management and use of active catalog components. It also provides a natural framework for advising users. Users who present expensive queries can be asked for specific attribute values to constrain the search for responses and lower processing cost.

An active catalog, meta-data caching, and referral graphs are currently used in a prototype descriptive name service called *Nomenclator*. Nomenclator answers selection and projection queries on relations that span heterogeneous name services in the global Internet. We use Nomenclator to evaluate our techniques for single-user and multi-user workloads. First, we show that Nomenclator improves descriptive query performance in a real name space, i.e. X.500, by constraining the search. Second, we show that Nomenclator uses a small investment of network bandwidth and server resources to improve the response time for a wide range of query sizes on a local area network. Last, we show that Nomenclator can amortize this investment over many queries to provide an overall reduction in network and server load, and, as a consequence, better scaling and response time for multi-user workloads.

This thesis does not address the issues of updates, security, and schema control in descriptive name services. Although name services rarely require large-scale updates, one relational interface for updating collections of component name services would be useful. Administrators could maintain replicas across several name services, making their naming data available to different user communities. Control over updates would allow Nomenclator to provide more accurate catalog functions and caches.

Security issues arise in several areas of our descriptive name service. First, the name service should not reveal information through its indexing structures. The one-way encryption of index structures or remote catalog functions are possible approaches to protecting privacy. Second, the name service must respect the security facilities of component name services. Although Nomenclator currently accesses only publicly available data, it can be extended to participate in the end-to-end authentication and encryption protocols supported by data repositories. Security constraints might reduce the utility of data caches, because fewer answers could be re-used. Meta-data caches, with properly encrypted or remote indexing structures, would not be affected. When the query is forwarded to a data repository, the data repository can return null answers to unauthorized users. Third, the creation of relations and addition of data to relations must be governed by access control. A first step would be to control what information can be added to the active catalog by organizational affiliation. Controlling the schema for a relation, which includes creating a relation and adding data repositories to a relation, is currently a manual process that should be automated, and we point out automation opportunities in the thesis.

1.3. Thesis Organization

This thesis is organized as follows. Chapter 2 reviews related work in name services and databases. It describes the challenges of scale, autonomy, and availability faced by information services in large internet environments.

Chapter 3 describes the active catalog. Components of the active catalog constrain the search space for queries and translate between heterogeneous environments.

Chapter 4 defines a referral graph and explains how the graph guides the creation, management and use of the active catalog.

Chapter 5 provides a detailed query processing example. It describes how the referral graph guides query processing, and it illustrates the contributions of meta-data and data caching to query performance.

Chapter 6 describes techniques for maintaining the active catalog. It explains how the referral graph directs the addition of components to the active catalog.

Chapter 7 presents our measurement and modeling results for single-user and multi-user workloads. We evaluate the performance of our techniques using network experiments, and an analytical model of very large internet environments.

Chapter 8 provides a summary and outlines some topics for future work.

Chapter 2

RELATED WORK

Descriptive name services provide an additional layer of naming over the resolution techniques that translate unique host names to network addresses [81, 118]. Historically, name service researchers investigated the nature of names and their relationship to objects in distributed systems [45, 88, 101, 103, 115]. It is now recognized that a name is a linguistic entity that identifies an object or group of objects. A distributed name service resolves the names of objects into other information that assists in accessing the objects. There can be several layers of name resolution between a name and access to an object; the information that results from one layer of name resolution supplies names that are resolved by the next layer [88]. Descriptive name services resolve information about the attributes of an object (or a query) into other attribute information about the object (or a tuple). Some of the attributes returned in the answer to a descriptive query can be the unique host names supported by earlier services.

As the concept of name in distributed systems has grown to encompass attribute-based queries and address to include tuples of attribute values, it has been recognized that a descriptive name service is a type of very large, distributed database service [78]. Descriptive name services and other information retrieval services must overcome the challenges of scale, autonomy and availability to make global operation a reality [5, 10, 78, 119]. These challenges are problems relevant to database research as well as networking and distributed systems research. Earlier name to address resolution services are a simpler form of descriptive service that subvert some of these challenges; they can be organized hierarchically to provide simple, distributed management in the presence of scale, autonomy and unavailability. The next section provides an overview of the relationship between distributed descriptive name services and distributed database services. Subsequent sections discuss the challenges facing large-scale descriptive name services in the context of network, distributed systems and database research.

2.1. Distributed Database Services

Descriptive name services, and most current name services, differ from distributed database services by having a weaker model of consistency. The appropriateness of a weaker model of consistency in name services was recognized in the early 1980's in distributed systems research [8, 114] and database research [37], and has been developed further by Terry [111]. The traditional approach to consistency in database research [79] treats the database as the authority for information, views the database as a sequence of consistent states in time and transactions as transformations between those states, and maintains consistency in the presence of any data interdependencies. In this model for example, the database is an authority for bank balances. It starts in a consistent state and faithfully reflects bank deposits and withdrawals. Name services differ substantially from these characteristics of authority, consistent states in time, and interdependencies.

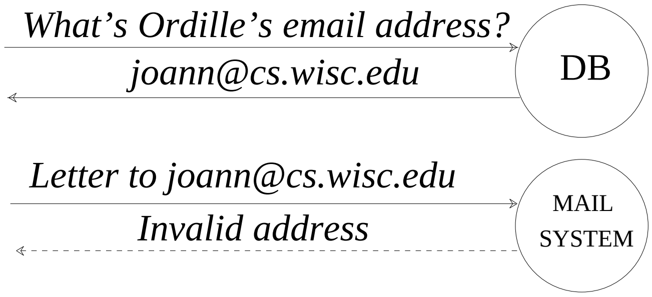


Figure 2.1. Names return hints

(The name service database supplies a hint. If the hint is invalid, the mail system notifies the user.)

In name services the object, not the database, is the authority for information. Name service responses are hints [111] that speed binding with the object. Users validate the hints when they contact the object, and recover if the hint is incorrect. Since serializability of query responses is so costly (if not impossible to obtain) in very large distributed name systems, systems with weaker consistency greatly improve performance by supplying hints that are seldom wrong and methods for recovering from the errors that do occur. For example, consider Figure 2.1. In this example a user asks the database for Ordille's electronic mail address. The user validates the electronic mail address received by sending mail to that destination. If the mail is not returned, then the address is valid. Events beyond the control of the naming system can cause name service information to be invalid, because object naming and binding are separate operations. For example, it is possible for a person's electronic mail address to change between the time of the name service query and the mail system's subsequent attempt to deliver a message. It is even more common for a person's address to change before the new address is recorded in the name service. Problems with invalid data can be reduced if external systems provide forwarding services to translate old binding information to new binding information. Cooper discusses other methods for identifying and recovering from invalid bindings [23]. In a system where the database is inconsistent with the world, worries about getting a true picture of a strictly consistent database seem pointless. Moreover, such a state is hard to define; problems of scale, autonomy, and availability make global synchronization impossible.

This is not say there is no sense of consistency in contemporary name services. Name services guarantee *tuple level consistency*, that a tuple will describe the state of an object at some time[1]. For example, an electronic mail address might be incorrect, but it was once the address of the person to be contacted. Weakly consistent

[1] Forwarding can be used to correct invalid hints, because each tuple is consistent.

t–bound queries have been suggested as a possible way to improve query performance in database systems [39]. A weakly consistent t–bound query returns the collection of tuples that satisfied the query at a time, s, greater than or equal to t. The collection is generated from a consistent state of the database at time s. Name service queries are weaker than weakly consistent t-bound queries, providing consistency at the level of a single tuple rather than a collection of tuples. Collections of tuples are not updated in a single transaction, because name service information has no interdependencies that require full transactions [8, 37, 114]. The creation, deletion, updating and reading of a tuple is atomic. A collection of tuples that is stored together on the same data repository is called a *partition* of the name space. Partitions are managed by a single administrator, and can be replicated on additional data repositories to increase availability. Replicas of partitions can diverge from the collection level consistency of transactions as long as tuple level consistency is maintained. When a tuple is updated in Grapevine [8], the system timestamps the change and propagates the tuple to its replicas. Two different replicas can update an attribute in the same tuple without knowing about each other's update; Grapevine uses the timestamp to identify the most recent change, which it then keeps. When tuples are read from many partitions in a name service, the collection of tuples represents information that may never have been current at the same time.

Some early research obscured the relationship between descriptive name services and distributed databases. White [118], Craft [25], and later Neufeld[72] defined descriptive names as being unambiguous, and, therefore, similar to the unique host names in earlier services. In this view, descriptive names are only valid if they identify a unique object [72, 118] or unique class of interchangeable objects [25]. Addressing mail messages and binding objects with descriptive names is simpler if the names are unambiguous. Although unambiguous descriptive names first appear to make descriptive name queries distinct from relational database queries, unambiguous descriptive names are a simpler, more constrained version of a relational selection predicate. The Profile Name Service describes unambiguous descriptive names as having an attribute that is a unique key in the name space [81], and Neufeld describes them as having a unique multi-attribute key [72]. While unambiguous descriptive names are simple to overlay on an existing hierarchical name space like X.500 [42, 72], they are a special instance of the more general and useful descriptive name that resolves to information about a collection of objects.

Profile [81] provides a descriptive query language based on preferences, a concept which has been further refined in Univers [12]. Descriptive queries can express preferences over the meta-characteristics of attributes. A user specifies general preferences for matching some types of attributes over others, because the attributes more reliably identify objects. For example, users will not find objects that have null values for the attributes specified in queries, so they prefer matches on attributes that are specified for every object over those attributes that contain null values. Users can also prefer matches for attributes that are true for the lifetime of an object over those that change, like processor load. The preferences identified by Profile can easily be expressed in the relational model. This requires that users know or can discover the meta-characteristics of attributes, like whether an attribute changes or can include null values. Once these characteristics are known, users can specify a series of relational queries that capture the preference ordering. Moreover, using the relational query language gives users the additional flexibility of choosing whether to apply the preference to some or all of the attributes in the query. A Profile interface could be built on top of a relational naming system, as it was built on top of the relational database in Univers [12].

The distinction between yellow pages and white pages name services also obscures the relationship between descriptive name services and distributed databases (compare for example [80] and [11]). Historically, the distinction refers to the domain of the objects named by descriptive services (services or people). Schwartz argues that the distinction between yellow and white pages services is that yellow page queries can be answered with information about a few of many available matching objects [92]. Allowing users to supply limits on the size of a result accomplishes this query for all types of descriptive name services. Efforts to optimize size limits could be made for those situations (network services) where a size limit is most likely to occur.

2.2. Scale

The scale of the global Internet is immense, both in number of objects and in geographic dispersion. There are now a million hosts on the Internet belonging to tens of thousands of organizations [63]. There are millions of Internet users distributed across every continent and terabytes of publicly available data [83]. Moreover, potential sources of information, like hosts and organizations, are growing exponentially.

There are two primary responses to locating objects in this information explosion. The first response seeks to organize information into a graph for ease of retrieval. Users direct the search for information by browsing through the graph. Graph-based services provide simple distributed management. They naturally incorporate data repositories distributed on the network by making them separate nodes or subgraphs in the global graph. Graph-based services, however, can be difficult to use. If users do not know the paths through the graph to the objects they seek, they can become lost and never find the desired resource. The second response seeks to provide non-procedural access to information wherever it is located. In successful systems, users do not direct the search; they provide descriptions of the information to be retrieved and the system directs the search. Descriptive name services and information retrieval systems are both examples of this type of service. Although these services relieve users of the need to understand the structure of a graph, they face performance challenges when information is distributed on a large scale. As one paper suggests [7], graph-based organization and descriptive services are complimentary not mutually exclusive.

2.2.1. Graph-Based Services

There are three categories of distributed graph-based services: file systems, hierarchical name services, and hypertext services. Distributed file systems and hierarchical name services are similar, because they support hierarchical pathnames, such as UNIX file names [85]. Levy and Silberschatz provide a survey of distributed file systems [59]. Alex [15], Prospero [73], Quicksilver [112] and the Stanford Name System [16] are large scale distributed file systems for Internet environments. Alex provides transparent file access to anonymous FTP directories, and supports caching of remote files with a validation (and update) frequency inversely proportional to the age of the file. Prospero transparently accesses remote file systems as well as anonymous FTP directories. Quicksilver reduces the communication latency in a geographically distributed file system by aggregating file operations. The Stanford Name System bridges the gap between local distributed file systems and global hierarchical name services. It incorporates multicast naming at the organization level into an existing global hierarchical name service such as Clearinghouse [75] or the Digital Network Architecture Naming Service (DNANS) [24, 55].

Clearinghouse, DNANS, the Domain Name Service (DNS) [66-68], and X.500 [22, 46, 102] use the decentralized hierarchical name space organization pioneered by Grapevine [8, 90]. Hierarchical name services and distributed file systems can delegate the authority to create and manage names for subtrees of the name space to different administrators in the network. Organizations have control of their naming data, and the effort to manage the data resides with those most interested in its availability. Hierarchical naming solved the problems of central name management that were experienced, for example, when the Network Information Center maintained a central table of hosts and addresses [69]. The Stanford Name System tailors the performance, security and reliability characteristics of different levels of the hierarchical name space to the differing needs of those levels. For example, multicast is used to resolve names in the local organization subtree, but other techniques are used to resolve names that cross the global root of the name space.

Hierarchical services are primarily designed to translate a unique name to other information about an object. The unique name of an object is a static set of attributes about the object that are encapsulated in the names of each level of the hierarchy. For example, the name `gruyere.cs.wisc.edu.` in the DNS describes a host named "gruyere" in the Computer Sciences Department of an educational organization called the "University of Wisconsin." If the set of attributes required by the unique name matches the set of attributes known by a user, translation is efficient. If a user needs to translate a totally different set of attributes, like the network address for "gruyere" into its name, it is necessary to search the entire name space or, as DNS does, build another tree that encapsulates this set of attributes. The need to translate different sets of attributes has recently led to a proposal to extend the DNS name space to include other translation trees [70]. Since hierarchical naming encapsulates attribute values in the naming tree, it is a special case of descriptive naming for a fixed set of attributes. The fixed set of attributes provide a unique multi-attribute key to the name space.

Decentralized control of the subtrees of the name space leads to a non-uniform name space structure. In much the same way that different users structure their file directory trees differently, different organizations vary in structuring their name spaces. Some organizations create subtrees for divisions and then departments; some create geographical and then department subtrees. When you add this complexity to the different kinds of subtrees needed to resolve queries on different combinations of attributes, the structure of the name space becomes too complex to locate anything quickly unless you already know its exact name. Several efforts have been made to reduce this complexity by allowing users to have personal naming trees [20, 21, 73, 82]. Once users identify objects of interest, they can organize the attributes of the objects into a hierarchical name space that they understand and find simple to use. These approaches provide a way of structuring and remembering information about objects once they are found, but they do not reduce the initial difficulty of finding the object, nor do they help you when the structure changes.

Terry [109, 110] points out that it is the requirement to distribute data according to the syntax of the name (by subtree) that creates some of the irregularity of the name space. When a subtree in the name space becomes too large for one server, it must be divided. This division can lead to the creation of rather arbitrary subtrees for the purpose of data management, for example the ever-present `/network-host` directories in distributed file systems. Terry recommends making name distribution independent of the structure of a name. Names can be assigned to servers by a function that divides names into groups using syntactic patterns, hashing or some other algorithm; a sequence of functions can also be applied to divide the names into successively smaller and smaller groups. These functions are not all globally known, but are discovered by moving from one server to another at the direction of a previous function call. Name resolution follows the sequence of functions to the location of the name. Like other hierarchical approaches, this approach distributes and then locates names as a function of the small set of attributes in the hierarchical name.

Several file system studies have indicated that hierarchical file name resolution is an important cost in system operation [38, 58, 71, 114]. In the absence of caching, these studies report that name resolution represents at least 40% of the system call overhead and 50% of the disk or network traffic. Fortunately, small caches provide high hit rates for hierarchical file name resolution and do not produce high processing or communication costs for maintaining cache consistency. In a recent trace-driven simulation study, a workstation cache of 10 whole directories produced a hit rate of 91% in the Sprite distributed file system [100]. A cache of 10 directory entries fared worse with a hit rate of 83%. The higher hit rate for the whole directory cache indicates that there is locality of reference to directories in a file system. This locality justifies the use of file name prefix caching in Unix [117] and the Stanford Name System [16]. In prefix caching, the location of directories is cached but the location of non-directory files is not. Mann and Cheriton report cache hit rates in excess of 99% for process-level prefix caches in the Stanford Name System [16]. Distributed file system studies also substantiate the relaxed consistency model by showing that cached file names are infrequently invalidated. Only 0.05% of the file names were invalidated during a simulation of 10 hours of host-level caching in LOCUS [98]. Approximately 0.25% were invalidated during a simulation of 24 hours of workstation-level caching in Sprite [100]. Mann and Cheriton report a smaller invalidation rate for prefix caches in the Stanford Name System. All misses (including invalidations) during 24 days of continuous operation of their system constituted 0.3% of the name resolutions.

Fewer performance studies have been done of hierarchical name services that are not also file systems. A recent study of X.500 hierarchical name resolution reports less temporal locality of reference for electronic mail addresses than previously reported for distributed files [9]. While the Sprite study produced a hit rate of 83% with a cache of 10 directory entries in a distributed file system, the X.500 study required a cache of 200 entries to attain a hit rate of 80%. Still, the X.500 hit rate is sufficiently high to corroborate the use of caching in hierarchical name services. Moreover, caching reduces the load on the network caused by remote name service requests. A DNS performance study identifies implementation and configuration errors that if eliminated would reduce DNS bandwidth usage by a factor of 20 [26]. The study provides some statistics on the effectiveness of caching for wide-area name service load reduction. The average number of hosts in an administrative domain that resubmit the same query in the two 24 hour periods of the study was 1.067 and 1.049 respectively. The small average number of repeating queries from the same domain indicates that caching is very effective in reducing wide-area traffic, especially because implementation or configuration errors in some domains produced 600 or more identical queries that are included in the average. The study also reports that traffic to update replicas of name space partitions was an insignificant percentage of the DNS network traffic.

Hypertext approaches to organizing global information provide a graph of objects where each object can embed pointers to other objects. The World-Wide Web [7], Information Mesh [105], and Distributed HyperText (DHT) [74] are large scale distributed hypertext services for Internet environments. Gopher[3, 28] also provides a graph of objects, but it does not provide the multi-media support and embedded pointers characteristic of hypertext approaches. It is simple to integrate new objects into a hypertext service by adding new nodes to the graph.

The difficulty in using hypertext services is sometimes called being "lost in hyperspace." Hypertext services are explicitly designed as browsing services. They do not provide unique names that users can remember to access useful objects again. Hypertext services do allow users to create subgraphs that contain pointers to interesting objects they have found in the larger graph [7]. In the absence of unique names, hypertext services have added two different kinds of descriptive services for locating objects. The first kind of descriptive service provides a hypertext node that dynamically creates a document with pointers to the objects that satisfy a descriptive query [7, 74]. The query results contain objects from a small part of the hypertext graph (e.g. objects from one host or organization) for which an index has been constructed. The other kind of descriptive service navigates some portion of the graph looking for nodes that satisfy a descriptive query [18]. Navigational search or partial indexing are expensive and incomplete solutions, respectively, to the problem of distributed descriptive name service performance.

2.2.2. Descriptive Services

Descriptive name services have typically lacked scale or function in large Internet environments. Several systems provide descriptive name services for hierarchical file systems [43, 71, 96, 112] or information about people [41, 44, 56], but do not provide large scale distributed access. The level of distribution permitted for descriptive queries in Profile [81], Univers [12], X.500 [22], Telesophy [89], and the Wide Area Information Server (WAIS) [28, 48] is limited by the number of servers it is feasible to contact. Profile and Univers support distributed access to chains of descriptive name servers specified by the user or listed on the server. Implementations of descriptive queries for X.500 exhaustively search subtrees of the hierarchical name space for objects satisfying a query. Telesophy and the Wide Area Information Server (WAIS) provide descriptive queries for full-text retrieval of documents on a local system. Telesophy supports distributed access by routing queries to all servers on a local area network. Distributed access in WAIS is based on browsing; users select the server to access from a directory of available servers.

Distributed database services avoid exhaustive search techniques for a subset of descriptive queries. Distributed database administrators can specify a set of disjoint selection predicates to distribute tuples in a relation to different data repositories. Each data repository is associated with a different predicate, and tuples that are placed at a data repository satisfy the predicate for that data repository. The set of selection predicates and their corresponding data repositories is called the *partitioning criteria* of a relation. Distributed databases replicate [108] or cache [40, 61] the partitioning criteria of a relation. The partitioning criteria acts as an index for those selection queries that fall within its logical scope; selection predicates that logically imply the partitioning criterion for a data repository can be answered exclusively at that data repository. Queries that do not imply a partitioning criterion do not scale well, because they require exhaustive searches.

Other descriptive name services limit functionality and more fully support distributed access. Unambiguous descriptive names limit descriptive name services to designating attributes that form a multi-attribute key to objects [72]. Netfind locates people by using information from USENET messages to direct `finger` queries to the most likely hosts for particular users on the Internet. Knowbots are processes that traverse the network searching for information of interest to their creators [49]. The heuristic searches used by Netfind and knowbots have a more limited semantics than full descriptive query support, because they can produce incomplete answers and provide little context for understanding a failed query.

Archie [35, 95] locates files on a network by making periodic exhaustive listings of files on hosts with anonymous FTP directories. The functionality of the supported descriptive queries is limited to regular expression matching of file names and a small set of keywords. The scale of Archie is limited by the same problems that led to the distribution of the DNS [69], viz. the Archie database is centralized and is limited in the scale of information it can supply by the resources available on one system. Replication of the database at 13 sites has increased the numbers of users Archie supports.

While Archie requires queries to be sent to a central index server, the Content Routing System [97], Indie [27, 28], the WHOIS++ Mesh [31, 116] and Enterprise [11] route queries through a distributed index to locate information and limit the number of data repositories contacted. Distributing the index allows these systems to support larger indices than Archie. The Content Routing System structures its distributed index as a tree of index servers. The root server indexes a subset of the available attributes; lower-level servers index additional attributes. Users navigate through the index by supplying values for attributes that are indexed at each level. Indie and the WHOIS++ Mesh allow administrators to structure index servers to meet the needs of their organizations. Indie relies on a centralized directory of descriptions of index servers to route queries to the correct server. The WHOIS++ Mesh is a graph of index servers; it routes queries from any mesh server to other mesh servers. The draft design for the mesh does not indicate how administrators determine the best location or content for their index servers. Neither Indie nor the WHOIS++ Mesh guarantee that queries return complete answers. Enterprise links servers to each other in a series of graphs that represent all the servers in the system, the geographic proximity of servers to each other, and the special interests of the people listed in the servers. In the specific cases of geographic proximity and special interest groups, Enterprise provides complete query results without exhaustive searches. No measurements are available for the performance of these systems for large search spaces of data repositories or many users. If the number of servers contacted by a query (index servers + data repositories) is less than the number of data repositories contacted by an exhaustive search, these systems have the potential to improve the scaling of queries to many data repositories.

Caching is successful in improving the scale of hierarchical name services, but caching for descriptive name services has received little attention. Data caching allows the results of previous queries to be re-used to answer new queries. Finkelstein provides a method for determining if new queries can be answered completely from the results of past queries [36]. Alonso, Barbara, and Garcia-Molina discuss data caching alternatives for local area networks with multicast capabilities [2].

2.3. Autonomy

The organizations that share information in the global internetwork are autonomous. Autonomy raises concerns about privacy, the reliability of local name services, and the integration of heterogeneous protocols and data formats. Organizations protect their privacy by requiring that their data are only read or updated by authorized users. When querying an organization, users require that the organization, and not some imposter, answer the query. Authentication and encryption services, such as Kerberos [107], protect data from unauthorized access and users from imposters. As the wealth of information available on the network grows, the need to protect the activities of users from observation also grows. Data suppliers can learn information about users from the pattern of queries users send to data repositories. New techniques are needed to protect the privacy of activities on the network from unauthorized observation by suppliers of services.

A name service is a valuable organization resource that must be secured from tampering. To understand this, one need only imagine the disastrous affects of failure to translate host names into addresses on a network, or locate essential personnel in a time of crisis. Hierarchical name services typically provide a fire wall between local and global name resolution [16]. They integrate local name services into a global name space while preventing external problems from disrupting local service. Local name services have enough context information to resolve names in the local subtree even if higher parts of the naming tree are unavailable. The same techniques can be used to build a fire wall for descriptive services. The local descriptive name service must have enough information to access local data even if the rest of the name space is unavailable. Heterogeneous distributed database systems (or multidatabases)[62, 99] also integrate information from autonomous organizations, but do not provide a complete fire wall between organizations. External transactions can disrupt local execution by holding local locks or forcing aborts of local transactions. Some multidatabase research seeks to reduce the violation of autonomy that occurs when transactions are used to preserve serializability [14, 84, 106]. The stricter model of consistency in multidatabases currently prevents the creation of the fire wall needed to protect local name services from external disruption. The relaxed model of consistency in name services permits better isolation of local processing.

Organizations can more easily share information if sharing does not require excessive software modification and support. Heterogeneous services reduce the software changes needed for global access by integrating existing protocols and data formats. Several meta-level name services provide one user interface to a collection of (component) name services. They encapsulate the names in component systems without requiring changes to the structure of the names, and often provide protocol translation to and from the component systems. These services require the user to direct the search to the component name services by providing a context identifier [17, 91], or they search all component services for responses [33, 34]. Univers [12] takes a different view of heterogeneity. Rather than supply one interface to many component name services, Univers supplies many interfaces to one data repository. Users continue to access naming data through whatever interface is comfortable to them. The managers of name services centralize naming data for ease of management.

Multidatabases provide techniques for integrating attribute and data conflicts [6, 53, 65]. For example, attribute conflicts occur when attributes that represent the same information disagree in name or type. Names are mapped to each other, and types converted. Data conflicts occur when data is contradictory or measured in different units. Contradictions are referred to administrators for resolution, and different units of measure are converted. Recent name service research also addresses data conflicts [11, 94]. Enterprise [11] chooses information from the most reliable source when contradictions occur. It also provides a callback mechanism to obtain updated information about users from the users themselves. Another system allows users to explore the information space by first searching the highest quality information and then, if unsuccessful, searching information of lesser quality [94]. Wiederhold suggests the development of techniques to support the use of uncertain data, because it is unlikely that all data conflicts will be resolved in truly global systems [119].

2.4. Availability

Availability should perhaps be called "unavailability" when we refer to the global network. If the probability that a name space partition is available is .999, then the probability that 10,000 independent partitions are available simultaneously is .0000452 (or 24 minutes of availability per year). Only 45% of X.500 partitions in the United States were available at one time during experiments in 1991 [76]. Measurements of the DNS indicate that more than 10% of the partitions at educational institutions are not replicated, so the unavailability of a single system can make one of those partitions unavailable [26]. The reasons for unavailability are failure, abandonment, and intermittent availability. As scale grows, the chance of failure in at least one system or network component increases dramatically. Failure can sometimes be addressed by replication [29], but there is a surprising absence of replication in essential services. Name space partitions are often abandoned by their owners when interest in the application wanes. The owners do not update the global catalog of partitions, and users continue to query the partition but are never answered. Organizations often make partitions unavailable intermittently due to maintenance schedules or privacy concerns. We have a slightly pessimistic, but not unrealistic rule of thumb: at any given time, at least some of the partitions that you wish to query will be unavailable.

Chronic problems with availability are incompatible with the strict model of consistency and traditional transaction processing of distributed database systems. If transactions require the availability of all name space partitions, transactions will never complete. The relaxed model of consistency for name services leads to algorithms that spread updates efficiently despite communications failures [30]. Incremental processing of queries complements techniques for increasing availability. A user can be given partial results for a query, and the option to have the query completed in the background when partitions become available. Automatic techniques for identifying and removing component systems that will never answer or have completely unreliable data are also important.

2.5. Summary

Descriptive name services augment earlier hierarchical name services. They relieve users of the need to understand the structure of the name space, a task that is increasingly difficult as the name space grows in size and heterogeneity. Descriptive name services are similar to distributed database services, but they differ by providing a relaxed model of consistency and processing a majority of selection and projection, rather than join, queries. The relaxed model of consistency enables name services to address the problems of scale with caching, autonomy with firewalls, and availability with replication. Hierarchical name services address scale by structuring the name space to provide fast query processing and distributed management of unique names. They use caching to provide good performance in the presence of large numbers of users. Descriptive name services have not adequately addressed scale in number of users or data repositories without curtailing functionality. Authentication services and firewalls preserve autonomy of users and organizations. Protecting the user from unwanted observation by information suppliers is an open issue for descriptive name services. The greater variety of information in autonomous descriptive services underlies a growing need for services that integrate heterogeneous data sources. Availability of name services is improved by replication. The large number of data repositories that may be contacted by a descriptive query exacerbates unavailability problems. Partial query evaluation and error recovery are needed to address these unavailability problems.

A recent paper identifies massive replication of small indices (e.g. in Archie) or partitioned massive indices (e.g. in WAIS or distributed indices) as the current techniques for supporting large scale descriptive services on the Internet [10]. Our previously published work and the work in this thesis provides a middle ground between these two techniques [76, 77]. We provide massive replication of parts of indices through meta-data caching at the user's site. Indices from different organizations can be integrated into one (perhaps massive) index by our referral graph techniques, but only parts of this index are replicated in meta-data caches. Hierarchical name services, distributed file systems and distributed database systems all provide limited meta-data caching. The caching of directory information in hierarchical name systems and distributed file systems provide meta-data about the location of files in those directories. Distributed databases replicate or cache the partitioning criteria of a relation. The partitioning criteria are meta-data that constrain a subset of descriptive queries to a subset of available data repositories. Systems that route queries through a distributed index have not emphasized meta-data caching at the user site, although the WHOIS++ Mesh allows for caching by returning referrals to the next step in the index traversal.

Our distributed indexing and caching techniques are more general than previous techniques. They provide and cache indexing information for any attributes, not just those in the partitioning criteria of a relation or the pathname of a directory. Index servers in a distributed index roughly correspond to catalog functions in our work. Catalog functions promote scaling to many data repositories by constraining the number of data repositories contacted for query results. Meta-data caches promote scaling to many users by decreasing the load on index servers and the wide-area network. We provide techniques for the description and algebraic manipulation of meta-data cache entries. Our techniques also structure graphs of catalog functions (or index servers) so that queries return complete results. Our prototype descriptive name service employs data caching techniques, and we provide a uniform framework for meta-data and data caching. We also address some issues in autonomy and availability. We use a simple technique to integrate data from different sources, so our prototype descriptive name service is independent of the heterogeneous data formats and protocols in the name space. We enhance the relaxed model of consistency to provide contextual information for interpreting partial query results.

Chapter 3

ACTIVE CATALOG

Users cannot afford to wait for a name service query to search thousands of data repositories when as few as 1% (or even 0%) of the sites hold the needed information. System capacity increases if we avoid unnecessary communication with data repositories. We introduce the *active catalog* as a mechanism to isolate queries within a subset of the data repositories that store a relation. The active catalog constrains the search space for a query, eliminating the overhead of contacting data repositories that do not contribute to the query answer.

In this chapter, we provide a preliminary description of the active catalog components and a system that uses the catalog (Section 3.1). We describe how information in the active catalog is structured into *access functions* that locate and retrieve information in the global name space. Access functions use syntactic, semantic, and data distribution information to limit the search space for queries (Section 3.2). They encapsulate the heterogeneous structures and access methods of the name space, making query processing independent of the proliferation of network name services (Section 3.3). We conclude this chapter with a description of the model of consistency supported by the active catalog.

In subsequent chapters, we expand the discussion to include methods for using, maintaining and building the active catalog. Chapter 4 describes the other major active catalog components, called *referrals*, that guide the use of access functions. Chapter 5 provides a detailed query processing example, and explains how the model of consistency is realized during query processing. Chapter 6 explains how the active catalog can be extended to new name services by adding access functions.

3.1. Preliminaries

The active catalog provides one relational interface to a world of heterogeneous data repositories. It structures the global name space into a collection of relations about people, hosts, organizations, services, and other resources. Users query these relations; they are freed from the need to understand the proliferation of heterogeneous services. The active catalog supports relations by collecting and organizing meta-data about the name space into *schemas*. Each relation schema defines the names and attributes of the relation. The schema also includes a list of referrals that describe the access functions used to locate and retrieve data in the relation. Access functions can be imported, used, cached, and re-used to answer queries at user sites, because the active catalog provides referrals that describe the conditions for using access functions. Referrals are described further in Chapter 4.

Access functions locate and retrieve the data in a relation. They respond to the question: "Where is data to answer this query?" There are two types of responses corresponding to the two types of access functions. The first type of 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 referral to data access functions that will answer the query or to additional catalog functions to contact for more detailed information. The second response to "Where?" is: "Here it is!" *Data access functions* return this response; they understand how to obtain query answers from specific data repositories. They return tuples that answer the query. We supply access functions for common name services, and organizations can write and supply access functions for their data.

Our prototype name service, Nomenclator, implements the active catalog using a distributed catalog service and a query resolver (see Figure 3.1). The distributed catalog service makes access functions and other meta-data from the active catalog available to users. Query resolvers at user sites retrieve, use, cache, and re-use this meta-data in answering user queries. Our catalog is *active*, because catalog functions generate additional meta-data that guide query processing. Typically, one resolver process serves several users on a local area network, so users can benefit from a larger resolver cache.

3.2. Catalog Functions

Catalog functions are central to the performance of our system. They provide an alternative to the exhaustive searches of many hierarchical name services, like X.500 [22], and are a generalization of data indices for a large internet environment. Catalog functions use many mechanisms to constrain the search space. For example, many name services, like X.500 or the Domain Name System [66], use organizational boundaries to partition the name space to different data repositories. If a query is covered by a particular organization, catalog functions can retrieve the description of the relevant physical partition from the underlying service. Catalog functions can also dynamically request information from local area networks that use broadcast service location facilities [1]. Catalog functions can improve performance by using semantic information, e.g. schema information about the legal locations for

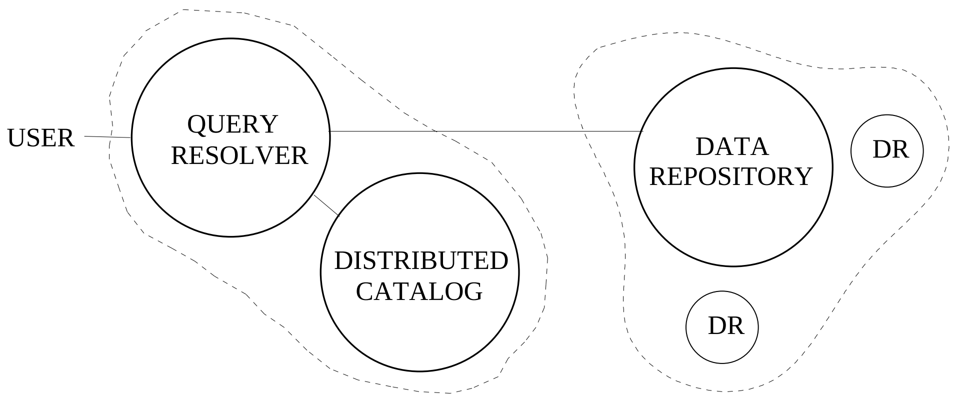


Figure 3.1. Nomenclator System Architecture

types of objects in the X.500 naming tree. Semantic information indicates which data repositories are incapable of storing information relevant to a query.

Catalog functions can also use data access functions to project the contents of a data repository onto an attribute and then use the result to build an index for the attribute. Sometimes catalog functions keep all the values for an attribute in an index data structure, but this is only feasible for a few attributes with a small number of values. More frequently, catalog functions use a variation of an index, called a *bit vector filter*, that has been used to improve the performance of distributed joins [4, 13, 32, 113]. Catalog functions hash each value of an attribute in a data repository, and then set the corresponding bits in the bit vector filter (see Figure 3.2). The catalog function hashes the value for the same attribute in a query, and then checks to see if the corresponding bit is set in the filter. The data repository is included in the search space only if the bit is set. If the bit is not set, the data repository is eliminated from the search. We used bit vector filters in a pilot study to achieve up to 40 times the performance of a common implementation of descriptive queries for X.500 [76]. Catalog functions extend the use of bit vector filters from their applications in distributed join optimization to the optimization of distributed selection predicates in large internets.

Catalog functions are implemented as remote or local services. Remote catalog functions are services that are available through a standard remote procedure call interface. Local catalog functions, as well as data access functions, are C [50] sources that are obtained by the query resolver from the distributed active catalog. The resolver dynamically compiles and loads them into its address space using an approach similar to CLAM [19]. Remote catalog functions are preferred over local ones when the resources of the query resolver are inadequate to support the catalog function. The owners of data may choose to supply remote catalog functions for privacy reasons if their catalog functions use proprietary information or algorithms. Remote catalog functions are 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, for example, when the function needs access to other

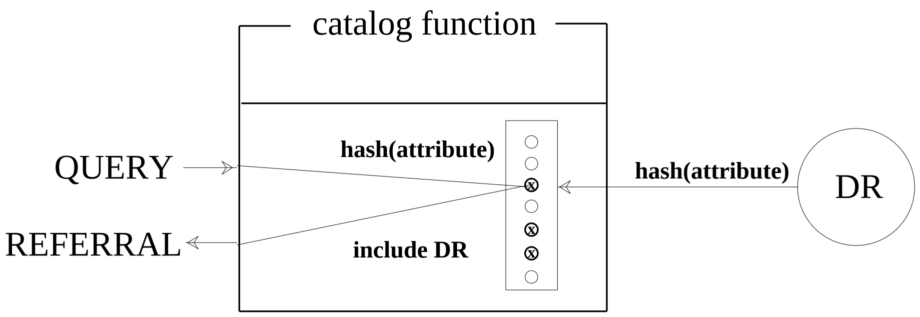


Figure 3.2. A catalog function with a bit vector filter

centrally located data sources. Local catalog functions are preferred whenever possible, because they are highly replicated in resolver caches. They can reduce system and network load by bringing the indexing resources of the active catalog directly to the users.

3.3. Data Access Functions

Data access functions encapsulate the heterogeneity of data repositories, making the query resolver independent of vagaries of different name services. Data access functions translate descriptive queries into queries that are understood by the data repositories and return tuples satisfying the descriptive query. Two types of mappings between descriptive queries and the data repositories make answering queries in a heterogeneous environment possible. First, data access functions map the attributes in the active catalog schema to their corresponding representation in the data repository. The attribute names and vocabularies must be translated to a common base by the functions. Second, data access functions map the relational operations of selection and projection to the corresponding operations in the data repository.

Nomenclator performs a straightforward, non-procedural mapping between its descriptive queries and other descriptive or relational name services. Mapping Nomenclator to a hierarchical name service is less straightforward. Data access functions for X.500 map attributes in one Nomenclator schema to attributes in a variety of X.500 objects. These functions combine attributes from different levels of the X.500 tree into one tuple in a Nomenclator relation. For example, the `people` relation, used in our X.500 experiments, 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. When attributes from different levels of the tree are used, the data access functions map descriptive queries to a combination of hierarchical navigation and descriptive search in X.500. For example, after a descriptive search retrieves a `person` with a required `commonName` attribute, hierarchical navigation retrieves the `countryName` attribute from a object superior to the `person` object in the X.500 tree.

3.4. Consistency Considerations

The relaxed model of consistency allows name services to address problems of scale, autonomy and unavailability in the name space. Greater scale is achieved, because updates are free of global synchronization bottlenecks and can propagate in batches when the system load is low. Autonomy is preserved, because organizations need not cooperate in update transactions. Unavailability is surmounted, because freedom from global transactions allows processing to continue when parts of the name space are unavailable. The relaxed model of consistency, however, does not explain the consistency of query responses to users. Users ask the following kinds of questions. When an answer is inadequate, how do you find a better one? When two organizations have different information about the same object, who is right? When data repositories are unavailable, how do you complete a query? The active catalog supplies advice that name services can use to answer these questions. *Advice* is information about the characteristics of a query or its answer. It provides a context for interpreting the response to a query, obtaining a better response, or constructing faster queries. For example, advice includes information about the currency or reliability of data in a response, and suggestions about which attributes can be added to a query to improve response time performance. We discuss currency and reliability advice in this section, and performance advice in Chapter 4.

The active catalog provides complete query responses. All information added to data repositories in the active catalog through the collection modification time (see `Collection Last Modified` in Figure 3.3) is searched for query responses. The collection modification time, or *t-bound* [39], is advice about the consistency of the response to a query. Tuples added after the collection modification time might not be reflected in the response. Each tuple is internally consistent, but may have been updated after the collection modification time. When users suspect that information is missing from their answer, they can request a more current collection of tuples. Since some catalog function indices are updated periodically, the name service can queue requests for more current answers until new active catalog information is available.

Consistency advice provides a context for interpreting query responses that is often lacking in other name services. Some services, like Netfind [93], sample the name space; they use heuristic searches that can miss searching relevant data repositories. Other services, like Profile [81], require users to direct the search by supplying the addresses of the data repositories to be contacted. Both kinds of systems generate incomplete answers. Incomplete answers from these systems are frustrating, because users have no recourse for obtaining missing information. The active catalog supports users in finding missing information, because users can request answers that are more current than the collection modification time of their last response.

The active catalog also supplies reliability advice in query answers. Data access functions support attributes that indicate the source of data in the tuple (i.e. the `source` attribute), and the time when the data was last changed (i.e. the `last modified` attribute). When conflicting information is received about the same object, as in Figure 3.3, the tuple modification time is a hint about which information is more likely to be correct. The most current update is not always the correct one, so other measures of reliability are also helpful. A consumer rating service could provide a reliability measure by recording the amount of incorrect data discovered for each `source` attribute

Name	Work Phone	Source	Last Modified
Ordille	608-262-6609	CsNet@wisc.edu	1-SEP-1992 17:22:41 CDT
Ordille	608-262-6617	WHOIS	7-JUN-1990 04:00:00 CST

`Collection Last Modified: 18-FEB-1993 03:30:11 CDT`

Figure 3.3. Query result with currency and reliability advice

value. The `source` attribute also helps users locate and report incorrect data to the administrator of that data.

Currency and reliability advice provides a context for deciding whether name service results (hints) are consistent with the world. It also makes it possible to process parts of queries in the background. If parts of the name space are unavailable, partial results can be given to the user and optionally the remainder of the results can be delivered when they become available. If a query takes too long, it can be completed in the background. The results are simple to interpret because they contain their own consistency information.

Descriptive queries have weaker consistency than weakly consistent t-bound queries [39], providing consistency at the level of a single tuple rather than a collection of tuples. Consistency and reliability advice differ from the types of consistency discussed for quasi copies [2] because the user discovers the inconsistency of the hint and demands more current information. In quasi copies, the system automatically updates caches to reflect a consistency requirement specified in advance by the user. In a name service environment, updating the user's information on demand consumes less system resources and is simpler for the user than specifying abstract consistency requirements.

Chapter 4

REFERRAL GRAPHS

Referrals are a general mechanism for describing distributed indexing structures. They direct query processing by stating the conditions for using access functions. Referrals form a DAG, called a *referral graph*, that integrates access functions into a uniform query processing framework. Referral graphs allow us to select the right access function for our needs and reap the benefit of multiple access functions in processing one query. Referral graphs direct the creation, management and use of access functions, so functions created by different organizations can constrain and access the name space to benefit everyone. They provide one mechanism for identifying meta-data or data cache entries that answer a query, thereby uniting our new meta-data caching techniques with existing data caching techniques.

Section 4.1 describes referrals and explains how referrals form a graph that guides query processing. Section 4.2 describes that relationship between catalog functions and referral graphs. It explains the techniques for generating referrals dynamically without storing the entire referral graph. We extend the referral format in Section 4.3 to provide additional opportunities for faster, more user-friendly query processing.

4.1. Referrals

Referrals govern the use of access functions. Each referral contains a template and a list of references to access functions (see Figure 4.1). The template is a conjunctive selection predicate that describes the scope of the access functions. Queries that are within the scope of the template can be answered with the referral. When a template contains a wildcard value ("*") for an attribute, the attribute must be present in any queries that are processed by the referral. Our system follows the following rule:

> *Query Coverage Rule:* if the set of tuples satisfying the selection predicate in a query is covered by ($\subseteq$) the set of tuples satisfying the template, then the query can be answered by the access functions in the reference list of the referral.

For example, the first referral in Figure 4.1 covers the queries in Figure 4.2.

Referrals can describe the partitioning criteria of a relation, and also can describe more complex indexing structures. For example, the `People` relation is partitioned by organization, and we can describe this partitioning criteria by a series of referrals like the second referral in Figure 4.1. This referral contains a data access function, and it covers the first query in Figure 4.2. Many distributed database systems, like Distributed INGRES [108], use an approach similar to ours for describing physical partitions of a relation, but the active catalog also does more than this by building indices for useful attributes, like a person's surname, which are not part of the partitioning criteria. The first referral in Figure 4.1 describes one such index. It contains a catalog function that returns a referral that

Template	Reference List
`c = "US" and name = "*"`	Catalog Function
`o = "UW"`	Data Access Function

Figure 4.1. Sample referrals for the `People` relation

```
1.  select * from People where
        name = "Miller" and
        o = "UW" and
        c = "US"

2.   select * from People where
        name = "Ordille" and
        c = "US"
```

Figure 4.2. Sample queries about people in organizations in the United States

lists the data repositories with information about people who have a particular surname in the United States. The wildcard in `name` indicates that the catalog function can constrain the search space for any `name` that is included in the query.

Referrals describe the tools (access functions) for locating and retrieving tuples, and the conditions (templates) for using those tools. They are the unit of meta-data caching in our system. Other systems, e.g. the Community Information System [40], Domain Name System [66], and R* [120], have simple versions of meta-data caching; these systems limit cached information about data distribution to the partitioning criteria of a relation. We achieve additional performance improvements by extending the information kept in the meta-data cache to index any attribute. Our referrals describe indices that span the entire relation, like the partitioning criteria, or describe indices that locate tuples for some other subset of the relation, like the catalog function in Figure 4.1.

Referrals form a generalization/specialization graph for a relation called a *referral graph*. Referral graphs are a conceptual tool that guides the integration of different catalog functions into our system and that supplies a basis for catalog function construction and query processing. A *referral graph* is a partial ordering of the referrals for a relation. It is constructed using the subset/superset relationship: $S \subset G$. A referral `S` is a subset of referral `G` if the set of queries covered by the template of `S` is a subset of the set of queries covered by the template of `G`. `S` is considered a *more specific* referral than `G`; `G` is considered a *more general* referral than `S`. For example, the subset relationship exists between the pairs of referrals with the templates listed below:

```
(1) (c = "US" and name = "Ordille") ⊂ (c="US")
(2) (c = "US" and name = "Ordille") ⊂
    (c="US" and name = "*")
(3) (c = "US" and name = "*") ⊂ (c="US")
(4) (c = "US") ⊂ ()
```

but it does not exist between the pairs of referrals with the following templates:

```
(5) (c = "US"), (dept = "CS")
(6) (c = "US" and name = "Ordille"),
    (c = "US" and name = "Miller")
```

In Lines (1) and (2), the more general referral covers more queries, because it covers queries that list different values for `name`. In Line (3), the more general referral covers more queries, because it covers queries that do not constrain `name` to a value. In Line (4), the specific referral covers only those queries that constrain the country to "US" while the empty template covers all queries.

During query processing, wildcards in a template are replaced with the value of the corresponding attribute in the query. For any query covered by two referrals `S` and `G` such that $S \subset G$, the set of tuples satisfying the template in `S` is covered by ($\subseteq$) the set of tuples satisfying the template in `G`. `S` is used to process the query, because it provides the more constrained (and faster) search space. The referral `S` has a more constrained logical search space than `G`, because the set of tuples in the scope of `S` is no larger, and often smaller, than the set in the scope of `G`. Moreover, `S` has a more constrained physical search space than `G`, because the data repositories that must contacted for answers to `S` must also be contacted for answers to `G`, but additional data repositories may need to be contacted to answer `G`.

Part of the referral graph for the `People` relation is shown in Figure 4.3. This example contains only referrals to data access functions. For simplicity of presentation, we leave out the data access function identifiers and list only the identifiers of the data repositories contacted by the data access functions. For example, referral `R1` lists the data repositories numbered from 22 to 54. The arcs in the graph indicate the path from a general referral to a more specific referral. Notice that referrals `R1` and `R2` are ordered from general to specific, but that `R1` and `R3` are not ordered by the graph. The direction of the arcs also indicates the direction in which the search space is

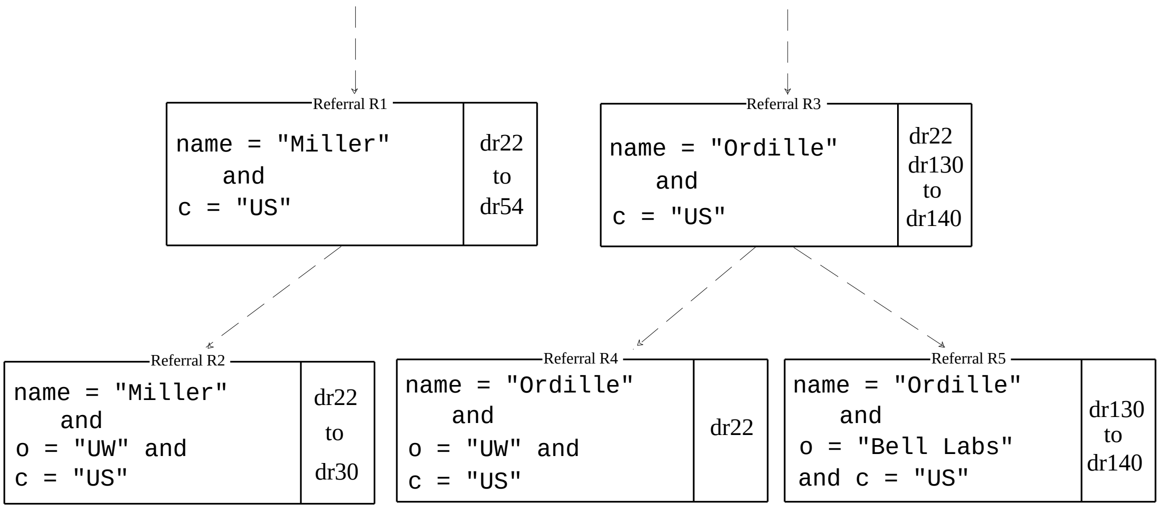


Figure 4.3. Sample partial referral graph

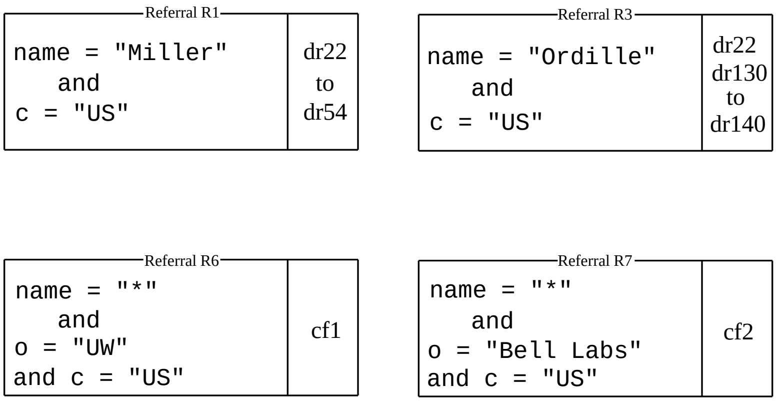


Figure 4.4. Catalog functions encapsulate parts of a referral graph

constrained. The first query in Figure 4.2 is covered by referral `R2` and also by referral `R1`, but it is answered using `R2`, the more constrained (and faster) referral.

The resolver query processing algorithm (see Chapter 5) navigates the referral graph, calling catalog functions as necessary to obtain referrals that narrow the search space. Sometimes, two referrals that cover the query have the relationship of general to specific to each other. The resolver eliminates unnecessary access function processing by using only the most specific referral along each path of the referral graph. The search space for the query is initially set to all the data repositories in the relation. As the resolver receives referrals to only data access functions, it forms their intersection to constrain the search space. The intersection of the referrals includes only those data repositories listed in both referrals. Intersection combines independent paths through the referral graph to derive benefit from indices on different attributes.

4.2. Catalog Functions

Catalog functions encapsulate portions of the referral graph. For example, catalog function `cf1` in Figure 4.4 uses the attribute `name` to direct queries to data repositories at the University of Wisconsin. It encapsulates referral `R2` and `R4` (from Figure 4.3), i.e. it can generate referrals `R2`, `R4`, or another referral as appropriate. Similarly, the catalog function `cf2` encapsulates referral `R5` and other referrals about surnames at Bell Laboratories. In constraining a query, a catalog function always produces a referral that is more specific than the referral containing the catalog function. Wildcards ("*") in a template indicate which attribute values are used by the associated catalog function to generate a more specific referral. In other words, catalog functions always follow the rule:

> *Catalog Function Constrained Search Rule*: Given a referral `R` with a template `t` and a catalog function `cf`, and a query $q \subseteq t$, the result of using `cf` to process `q`, `cf(q)`, is a referral R' with template t' such that $q \subseteq t'$ and $R' \subset R$.

Catalog functions can also encapsulate other catalog functions by calling them. For example, we can replace the entire graph in Figure 4.4 with the referral in Figure 4.5. The catalog function `cf3` calls `cf1` and `cf2`, and returns the union of their results (i.e. a referral containing all the reference lists from the referrals returned by `cf1` and `cf2`). When catalog functions call other catalog functions (or return referrals to them), they form a DAG of catalog functions that is a more compact, functional representation of the referral graph. Catalog function DAGs perform hierarchical indexing on multiple attributes. Catalog functions at a root of a DAG, like `cf3`, use one or more attributes, in this case organization (`o`), to choose relevant localities in a large search space. They further reduce the search space by calling more specific catalog functions that are tailored to those localities, and form the union of their results.

4.3. Revised Templates

When a catalog function forms the union of multiple referrals, some specificity can be lost. For example, if we process the second query in Figure 4.2 using `cf3`, we receive a referral with the template `name =`

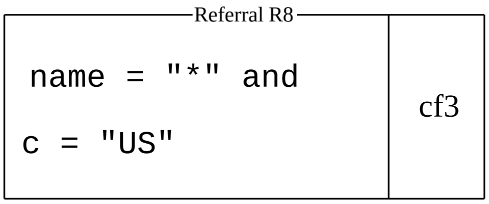
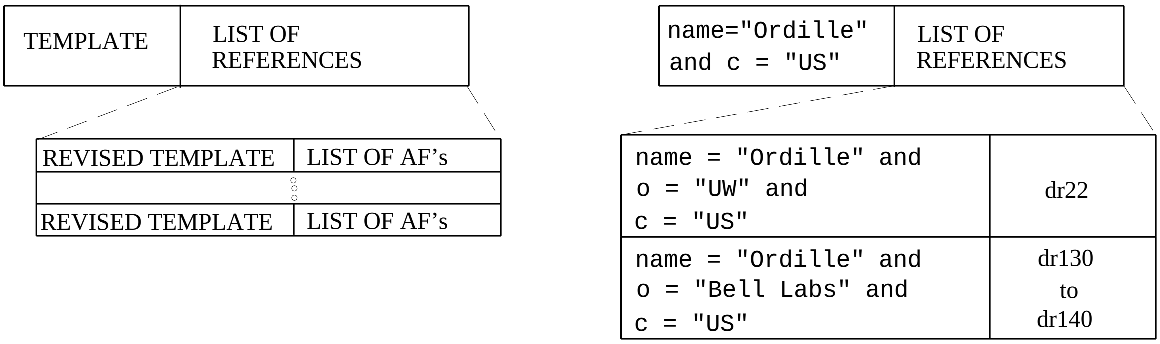


Figure 4.5. A catalog function encapsulates Figure 4.4

`"Ordille" and c = "US"`. This referral is the union of more specific referrals (from `cf1` and `cf2`) that contained the organization attribute in their templates, but we lose the organization information associated with parts of the search space when `cf1` constructs the union. We would like to have this more specific information, because it helps us to find previously cached subquery answers (in this case, a query for "Ordille" in a particular organization) and to advise users on how to add attributes to their queries to reduce the search space. To provide more specific information from the referral graph when using general catalog functions, we adopt the general referral format in Figure 4.6(a). Each referral can qualify its references to access functions with a revised template. The revised template follows the Query Coverage Rule with respect to its associated access functions. A catalog function uses the general format to collapse a section of the referral graph into one referral. For example, `cf3` can collapse the graph rooted at referral `R3` in Figure 4.3 to the referral in Figure 4.6(b). The resulting referral is the union



(a) General format **(b) An example**

Figure 4.6. Referrals

of leaves of the referral graph; its revised templates and access functions are the templates and access functions of the leaves.

The construction of general referrals with template `g` and revised templates `rt1`, `rt2`, ..., `rtn` follows two rules. The first rule is the following:

Referral Coverage Rule: $g \subseteq rt1 \cup rt2 \ldots \cup rtn$.

This rule, like the Query Coverage Rule, is required for correctness. Catalog functions forming the union of referrals must know that the union covers the scope of the returned template. For example, the catalog function `cf3` can only return the referral in Figure 4.6(b), because it has contacted every organization in the United States and found the locations where "Ordille" is listed. Catalog functions can use other information, such as knowledge of the structure of the name space, domains of attributes, or integrity constraints on tuples, to return a general referral with revised templates. The second rule is the

Referral Constrained Search Rule: $g \supseteq rt1 \cup rt2 \ldots \cup rtn$.

This rule, like the Catalog Function Constrained Search Rule, is true by construction, because catalog functions always walk the referral graph by adding attribute values to templates. The two rules for referral construction allow us to deduce that $g = rt1 \cup rt2 \ldots \cup rtn$. This equality is an empirical observation about the structure of the name space at a particular time. For example, Figure 4.6(b) asserts that the set of tuples satisfying `(name = "Ordille" and c = "US")` equals the union of the set of tuples satisfying `(name = "Ordille" and c = "US" and o = "UW")` and the set of tuples satisfying `(name = "Ordille" and c = "US" and o = "Bell Labs")`. More colloquially, it asserts that the only people named Ordille in the United States name space are at the University of Wisconsin or Bell Laboratories.

When a data access function is described by a revised template, the query resolver performs two optimizations. The intersection of the query and revised template is the subquery answered by the associated data access function and data repositories. If the answer to the subquery is in the data cache, the cached answer is used and the data repository is not contacted. If the subquery is logically inconsistent, the contents of the data repository contradict the query and the data repository is not contacted. For example, a subquery that includes the following conjunction is inconsistent: `o = "UW" and o = "Bell Labs"`. Referrals can also be used to add an *advice phase* to the query processing algorithm. When the final search space is too large to process quickly, users can optionally receive a list of attributes that would narrow the search further. For example, the resolver presents the attribute values in revised templates, but not in the query, to the user. The user can select attribute values from this list to constrain the query further.

Referrals and the four simple rules summarized in Table 4.1 allow us to unify a wide variety of indexing techniques. Catalog functions contributed by different organizations can be integrated into one structure to speed query processing for everyone. Referrals and the meta-data rules also unite our meta-data caching techniques with popular data caching techniques. Like other systems, Nomenclator uses techniques developed by Finkelstein[36] to cache and re-use the data responses to queries. Since both meta-data and data cache entries are tagged with selection predicates, the query resolver uses the same algorithm in either cache to determine if a cached entry covers a query. Our query processing techniques traverse referral graphs to constrain queries to specific search spaces. They allow us to reap the benefits of multiple indices by integrating the referrals from catalog functions for different parts of a relation into one referral, and by forming the intersection of referrals that cover the same query.

Rule Name	Rule Summary
Query Coverage Rule	**If** q $\subseteq$ t, **then use referral**
Catalog Function Constrained Search Rule	**If** R **contains** t **and** cf, q $\subseteq$ t, **and** cf(q) **returns** R′ **with template** t′, **then** q $\subseteq$ t′ **and** R′ $\subset$ R
Referral Coverage Rule	g $\subseteq$ rt1 $\cup$ rt2 ... $\cup$ rtn
Referral Constrained Search Rule	g $\supseteq$ rt1 $\cup$ rt2 ... $\cup$ rtn

Table 4.1. Meta-data rules

(For a query q *; a referral* R *with template* t *and catalog function* cf *; a referral* R′ *with template* t′*; and a referral with template* g *and revised templates* rt1*,* rt2*, ...* rtn*.)*

Chapter 5

QUERY PROCESSING AND CACHING

In this chapter, we bring together our discussion of the active catalog and referral graphs in a detailed query processing example. Our examples show how to initialize a query resolver, process multiple referrals that cover the same query, generate referrals from catalog functions, re-use cached information, and benefit from revised templates. We describe the flow of query processing through the data cache, meta-data cache, distributed catalog service, and data repositories. Our initial example considers search spaces comprised of data repositories that hold distinct parts (partitions) of the name space. We expand this example to process search spaces with replicated partitions. Finally, we explain how the query resolver uses advice from the active catalog to support the consistency model of Chapter 3.

We limit our discussion to query processing techniques for conjunctive queries. Queries containing disjunctions can be processed in a similar manner by converting them to disjunctive normal form [60], determining the search spaces for the component conjunctions, and forming the union of these search spaces. Join processing in large Internet environments is an area of future research.

In Chapter 6, we conclude our presentation of techniques that support fast descriptive query processing by describing how to maintain the active catalog. Subsequent chapters present experimental and modeling studies of our techniques and our conclusions.

5.1. Query Processing Overview

Our prototype system, Nomenclator, includes many local query resolvers and a distributed catalog service as described in Chapter 3. The query resolver is shared by a group of users on the local area network, much like DNS servers are shared at the organization or department level. Access functions and referrals from the active catalog are replicated in query resolver meta-data caches to tailor the resolver to its stream of queries. Meta-data caching is a partial replication; no query resolver contains the entire contents of the active catalog but rather those parts that are currently most useful to it. New referrals are generated as needed from catalog functions. Referrals that match recurring queries at the resolver are cached and re-used. Referrals and access functions make the resolver's query processing algorithm independent of the access methods and search techniques in the name space.

Figure 5.1 shows the flow of processing through the query resolver. When a query arrives, the resolver checks the data cache for a previous response that covers the query. If the data cache misses, the resolver must determine the search space for the query. It checks the meta-data cache for previous referrals that cover the query. If none are found, the resolver contacts the distributed catalog service for an initial set of referrals for the query.

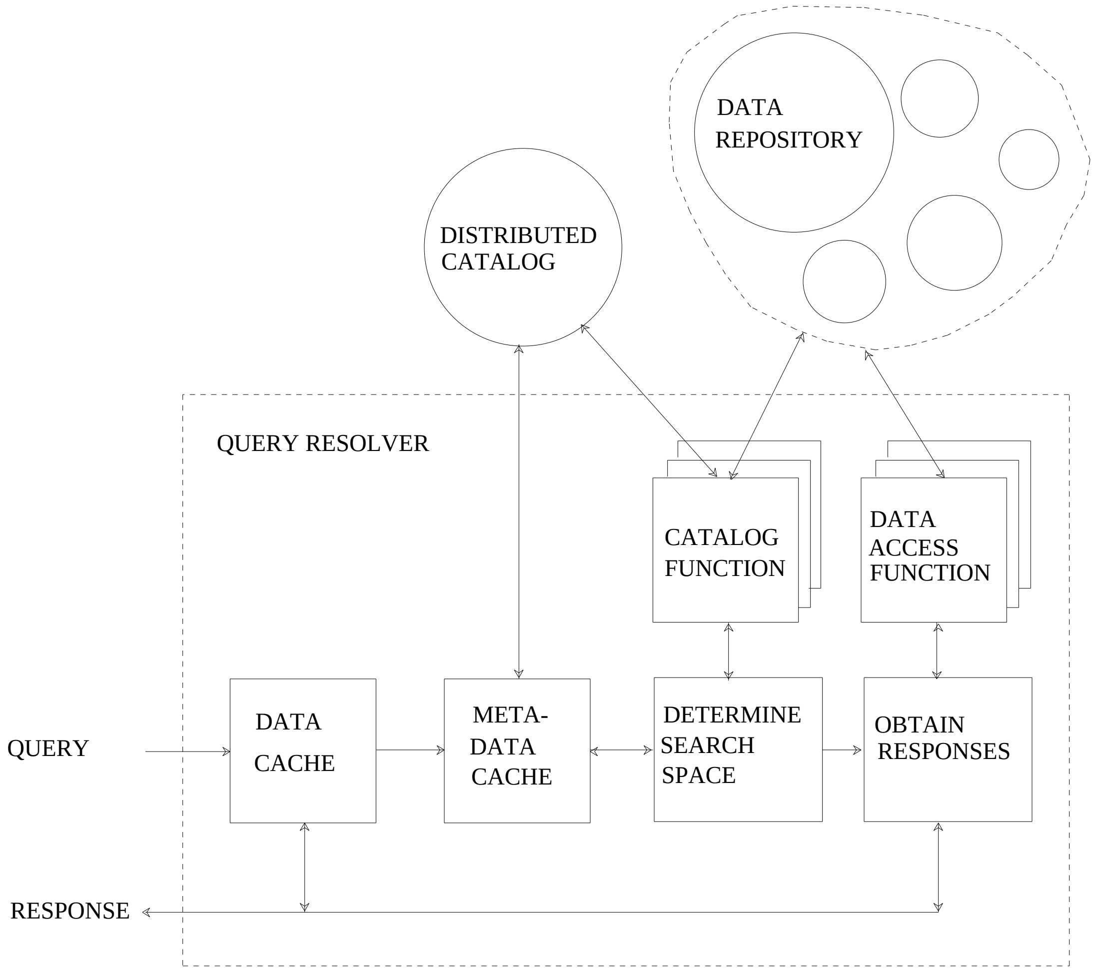


Figure 5.1. Query resolver processing

The resolver processes the initial set of referrals until it determines the search space for the query. The search space is created by forming the intersection of referrals that only reference data access functions. The resolver uses referrals that reference catalog functions to generate additional referrals to take part in the intersection. The intersection algorithm of the resolver is described in Section 5.2. The resolver caches the referrals generated by catalog functions in the meta-data cache.

The final search space for the query is a referral that only references data access functions. The resolver uses the data access functions to obtain query responses from each data repository listed in the referral. If the referral has revised templates, the query resolver may be able to avoid contacting some data repositories. It forms the intersection of the revised template and the query, and checks for answers to this subquery in the data cache before contacting the relevant data repository (not shown in Figure 5.1). The final result for the query is cached. During query processing, if a data access function or catalog function is not available locally, the query resolver obtains it from the distributed catalog service and places it in the meta-data cache.

5.2. Intersection Algorithm

A goal of the resolver is to constrain the search space to a small set of data repositories. Two types of referrals can lead to this constrained set. A *data referral* is a referral that only references data access functions; it lists a constrained set of data repositories to search. A *catalog referral* is a referral that includes references to catalog functions; the catalog functions must be called to obtain a list of data repositories. When multiple data referrals cover the query, the answer to the query can be found in the intersection of the physical search spaces (or sets of data repositories) defined by the referrals. Since intersection is a mechanism for reducing the size of the search space and combining the indexing information in multiple referrals, the resolver uses an intersection algorithm to process referrals.

Only data referrals can participate directly in the intersection algorithm of the resolver, because only data referrals enumerate the search space for the query. The intersection of two data referrals, `R` and `R′`, is a referral, `I`, that covers every query covered by both `R` and `R′`. Only tuples that satisfy the predicates in the templates of `R` and `R′` can satisfy the template of `I`, so the template of `I` is the conjunction of the templates in `R` and `R′`. Only tuples from physical locations listed in both `R` and `R′` can satisfy queries covered by `I`, so the set of data repositories listed in `I` is the intersection of the set of data repositories listed in `R` with the set listed in `R′`. Each data repository in `I` is listed with a data access function and optionally a revised template. When a revised template is present in `I`, it is the conjunction of a pair of revised templates that occurred in references to the data repository in `R` and `R′`. Table 5.3 provides an example of the intersection of two data referrals with revised templates. In Tables 5.3a and 5.3b, the intersection of Referral 6 with Referral 8 is Referral 9.

A catalog referral does not define a set of data repositories to search for query responses; therefore, it can not participate directly in the intersection of referrals by the resolver. The catalog functions in the referral must be called to obtain a data referral that covers the query. Typically, a referral to a catalog function contains one reference with a catalog function and no revised template. By the Catalog Function Constrained Search Rule (see Section 4.2), calling the catalog function in this referral produces a new, more specific referral that covers the query. If a catalog referral, `N`, contains multiple references, each catalog function in `N` returns a referral that covers only part of the search space for the query. The resolver generates a new, more specific referral from `N` by replacing parts of `N` with the results of calling the catalog functions listed in `N`. The reference list in the referral returned by a catalog function replaces the reference containing the catalog function in `N`. The template of `N` is updated to reflect the additional attributes in the query that are constrained in the template of the referral returned by the catalog function. If the new referral generated by calling catalog functions is a data referral, the new referral is available for intersection to constrain the physical search space of the query. If the new referral is a catalog referral, the process of calling catalog functions repeats.

```
Attribute Name (Abbreviation):  Sample Value

Given Name (gname):             Joann
Surname (name):                 Ordille
Email Address (email):          joann@cs.wisc.edu
Work Phone (wph):               (608) 262-6617
Department (dept):              Computer Sciences
Organization (o):               University of Wisconsin
City (city):                    Madison
State or Province (state):      WI
Country (c):                    USA
Last Modified (mod):            12-FEB-1991 22:05:14 CST
Source (s):                     X500:@c=US
                                @o=University of Wisconsin
                                @ou=Computer Sciences
```

Figure 5.2. Attribute names, abbreviations and sample values for the `People` relation

The resolver establishes an initial search space for the query by forming the intersection of the data referrals that cover the query. This intersection is a referral called SearchSpace. The resolver uses SearchSpace and the partial ordering defined by the referral graph to determine which of the catalog referrals that cover the query will further constrain the search. The resolver orders the catalog referrals from specific to general as specified in the referral graph. Any attribute listed in the template of SearchSpace has been used to constrain the query, so the template of a catalog referral must have an attribute, not in the template of SearchSpace, in order to constrain the query further. The ordering and the check against the SearchSpace template ensure that the resolver does not call a catalog function if it has the result of the catalog function available for use in processing the query. Catalog functions in a referral that will further constrain the search are called until a data referral results. This data referral is intersected with SearchSpace to produce a more constrained search. Once all catalog referrals have been considered, the resolver contacts the data repositories in SearchSpace to obtain the final result for the query. Several examples of resolver's query processing algorithm follow in Section 5.3.

5.3. Query Processing Example

As an example of query processing and meta-data caching, we describe the execution of four queries on the `People` relation. Figure 5.2 lists the attribute names in our example relation, their abbreviations, and sample attribute values. Table 5.1 lists the four queries to be processed. Queries 1 and 2 request information about people named respectively "Ordille" and "Miller" in Wisconsin in the United States. Queries 3 and 4 request information about people named "Ordille" in the United States who work respectively at the University of Wisconsin in Wisconsin or Bell Laboratories in New Jersey. The query resolver begins its processing with empty caches, and retrieves or generates a series of referrals to direct the search for query answers. These referrals are identified by number, and are listed with the identifier of the source that generated them in Tables 5.2 through 5.5. We simplified our

No.	Query

1	select * from People
	  where name = "Ordille" and
	    state = "WI" and c = "US"

2	select * from People
	  where name = "Miller" and
	    state = "WI" and c = "US"

3	select * from People
	  where name = "Ordille" and
	    o = "University of Wisconsin" and
	    state = "WI" and c = "US"

4	select * from People
	  where name = "Ordille" and
	    o = "Bell Labs" and
	    state = "NJ" and c = "US"

Table 5.1. Sample queries on the `People` relation

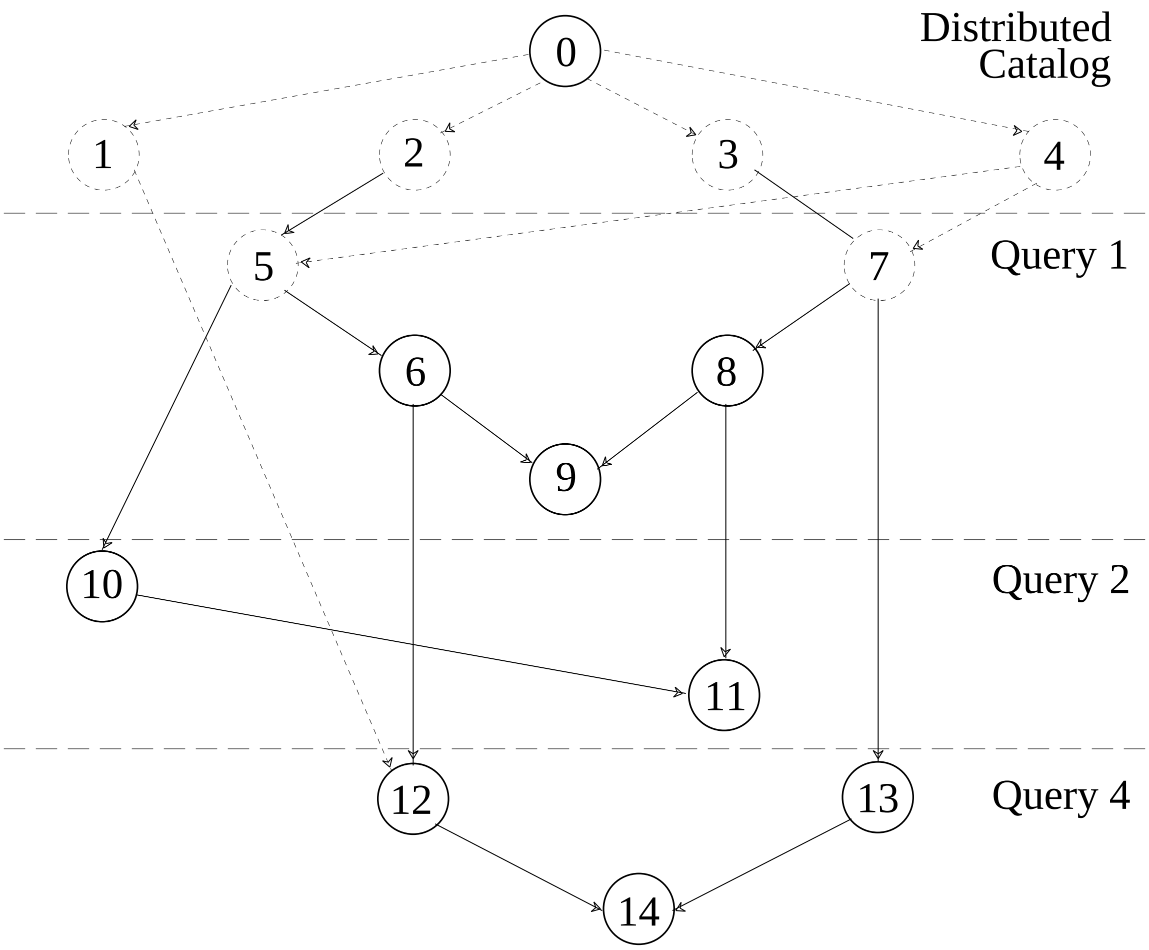


Figure 5.3. Referral graph generated by example queries

(Graph nodes contain the ID number of the referrals listed in Tables 5.2 through 5.5. Dotted-lined nodes represent catalog referrals. Bold-lined nodes represent data referrals. The arcs point from a general referral to a more specific referral. Bold-lined arcs also indicate that the general referral was used to create the more specific referral. Referrals appearing between parallel horizontal lines were added to the meta-data cache as a result of a call to the distributed catalog service or the processing of a query as indicated.)

referrals by including a smaller number of data repositories than would normally be listed in referrals in a large Internet environment. The graph for the referrals is provided in Figure 5.3.

5.3.1. Query 1

The processing of Query 1 provides an example of meta-data cache initialization, catalog referral processing, and data referral intersection. When Query 1 arrives, the resolver finds no information in either cache, so it contacts the distributed catalog service for referrals in the `People` relation. The distributed catalog returns the referrals in Table 5.2. Referral 0 is the *base referral*. The base referral is the most general referral for a relation; it provides data access functions for generating and contacting every data repository in the relation. The base referral is seldom used to contact data repositories, because the active catalog provides other more constrained referrals. Since the set of all data repositories is seldom needed, it is not enumerated in the base referral. The resolver processes the base referral to ensure that the intersection of any data referral, `R`, with the base referral is `R`; therefore, the base referral can be used to initialize the search space for a query even though the data repositories are not enumerated in the referral. Referrals 1 through 4 list the attributes that the active catalog can use to constrain the search; they provide entry points into the referral graph for the relation. Referrals 0 through 4 comprise the most general levels of the

Ref. ID	Source of Ref.	Template Revised Template	Access Function (Data Repositories)
0	DC		Everywhere_DAF (Everywhere)
1	DC	o = "*"	o_CF
2	DC	name = "*"	name_CF
3	DC	state = "*"	state_CF
4	DC	c = "*"	c_CF

Table 5.2. Referrals for the `People` relation from the distributed catalog

(Each referral is identified by number during the discussion of the query processing example. The source that generated the referral is "DC" for distributed catalog or the number of the referral containing the source catalog function. Revised templates are indented and listed parallel to their associated access functions. Access function names end in "CF" for catalog functions or "DAF" for data access functions. Data access functions are followed by the list of data repositories they contact.)

referral graph in Figure 5.3. The additional levels will be added during the processing of our example queries.

The resolver initializes SearchSpace to the only data referral that covers the query, Referral 0. Referrals 2 through 4 also cover Query 1 (see Tables 5.1 and 5.2), because they only index attributes that are present in the query. These catalog referrals form the ordered list of candidate referrals for processing: Candidates = {Referral 2, Referral 3, Referral 4}. Since no referral in Candidates is more specific than the others (as illustrated in Figure 5.3), the resolver processes them in the original order. When the query resolver calls the catalog function in Referral 2 to constrain the `name` attribute, it receives Referral 5 to a catalog function (`name_US_CF`) that can locate `name` in the United States. Calling `name_US_CF` produces Referral 6, a data referral with two revised templates. The revised template specific to the University of Wisconsin is associated with one data repository, and the one specific

Ref. ID	**Source of Ref.**	**Template / Revised Template**	**Access Function (Data Repositories)**
5	2	`name = "*" and c = "US"`	`name_US_CF`
6	5	`name = "Ordille" and c = "US"`	
		`o = "UW" and name = "Ordille" and c = "US"`	`X500_DAF (dr22)`
		`o = "Bell Labs" and name = "Ordille" and c = "US"`	`Unix_10_DAF (dr130 – dr140)`
7	3	`state = "*" and c = "US"`	`state_US_CF`

Table 5.3a. Referrals generated while processing Query 1

to Bell Labs is associated with 11 data repositories. The query resolver forms the intersection of Referral 6 with the search space (currently Referral 0) and obtains SearchSpace = Referral 6.

Referral 3 describes a catalog function that indexes the `state` attribute. Referral 3 and SearchSpace are not ordered by the referral graph, because `state` has not previously been used to constrain the search space. The resolver calls the catalog function in Referral 3 to further constrain the search space. It obtains Referral 7 to the catalog function `state_US_CF` that can locate `state` in the United States. Calling `state_US_CF` results in

Ref. ID	**Source of Ref.**	**Template / Revised Template**	**Access Function (Data Repositories)**
8	7	state = "WI" and c = "US"	X500_DAF (dr1 – dr30) CSNET_DAF (dr55 – dr75) WHOIS_DAF (dr76-99)
9	0 ∩ 6 ∩ 8	name = "Ordille" and state = "WI" and c = "US"	
		o = "UW" and name = "Ordille and state = "WI" and c = "US"	X500_DAF (dr22)

Table 5.3b. Referrals generated while processing Query 1 (continued)

Referral 8, a data referral for the state of Wisconsin in the United States. The intersection of Referral 8 and SearchSpace is Referral 9. Referral 9 is the new value of SearchSpace, and more specific than either Referral 8 or Referral 6 (the previous value of SearchSpace). The result of the intersection of two referrals that cover a query is always more specific or identical to the operands of the intersection. The last referral in Candidates, Referral 4, is more general than SearchSpace, so the catalog function in Referral 4 is not called. Referral 4 uses `country` to constrain the search space, and that constraint has been added to SearchSpace by previous intersections.

The resolver uses Referral 9 to obtain the response to Query 1. The resolver forms the conjunction of the query and the revised template in Referral 9 to obtain the subquery (identical to Query 3) answered at data repository `dr22`. After finding no response to this subquery in the data cache, the resolver uses data access function `X500_DAF` to obtain the response from `dr22`. The resolver returns this response to the user, and caches the query and the response in the data cache.

Ref. ID	**Source of Ref.**	**Template** / **Revised Template**	**Access Function (Data Repositories)**
10	5	`name = "Miller" and c = "US"`	
		`o = "UW" and name = "Miller" and c = "US"`	`X500_DAF (dr22 – dr30)`
		`o = "UC" and name = "Miller" and c = "US"`	`WHOIS_DAF (dr31-54)`

Table 5.4a. Referrals generated while processing Query 2

5.3.2. Query 2

The processing of Query 2 provides an example of re-using data and catalog referrals from the meta-data cache. It shows how the intersection algorithm ensures that referrals provided by the distributed catalog or catalog functions are always re-used before the distributed catalog or catalog functions are re-contacted.

The answer to Query 2 is not in the data cache. The resolver initializes the search space from the meta-data cache to the intersection of the data referrals that cover the query; therefore, the search space is the intersection of the base referral with the referral to data repositories for Wisconsin in the United States, i.e. SearchSpace = (Referral 0 ∩ Referral 8) = Referral 8. The catalog referrals that cover Query 2 are ordered in the list of candidates: Candidates = {Referral 5, Referral 7, Referral 2, Referral 3, Referral 4}. The candidates index `name` in the United States, `state` in the United States, `name`, `state`, and `country` respectively. The referrals are ordered by the number of attributes in their templates that have specific values (not wildcards) and then by the total number of attributes present. This ordering preserves the partial orderings from specific to general shown in the referral graph in Figure 5.3.

The resolver uses Referral 5 that indexes `name` in the United States to generate Referral 10, a data referral for `name = "Miller" and c = "US"`. The intersection of SearchSpace (i.e. Referral 8) and Referral 10 gives Referral 11. Referral 11 is the final SearchSpace for the query, because all the other candidate referrals are more general than Referral 11. Notice that the ordering of the list of candidates guaranteed that Referral 5 was used

Ref. ID	**Source of Ref.**	**Template** **Revised Template**	**Access Function (Data Repositories)**
11	0 ∩ 8 ∩ 10	name = "Miller" and state = "WI" and c = "US"	
		name = "Miller" and o = "UW" and state = "WI" and c = "US"	X500_DAF (dr22 – dr30)

Table 5.4b. Referrals generated while processing Query 2 (continued)

before Referral 2, the source of Referral 5. Generating the search space for Query 2 re-used three referrals obtained while processing Query 1 (Referrals 0, 5 and 8). It generated fewer referrals than Query 1, because it was able to re-use the cached results of calling catalog functions (Referrals 5 and 8) and contacting the distributed catalog service (Referral 0). The resolver uses Referral 11 to obtain the responses to Query 2.

5.3.3. Query 3

The processing of Query 3 provides an example of eliminating meta-data processing by finding a query answer in the data cache. Query 3 is covered by the selection predicate in Query 1, so resolver derives the answer to Query 3 from the data cache. The answer for Query 3 is identical to the answer for Query 1, because Query 3 is the subquery answered by `dr22` during the processing of Query 1.

5.3.4. Query 4

The processing of Query 4 provides an example of using a revised template to obtain a more constrained referral that covers the query. Query 4 is covered by the Data Referrals 0 and 6. The resolver initializes SearchSpace to Referral 0, and then performs an additional optimization step before using Referral 6 to constrain SearchSpace further. The references with revised templates in Referral 6 are parts of the referral graph that are more specific than Referral 6; these references are independent referrals that were collected together to answer queries with more general scopes than Query 4. Since Referral 6 was generated for a different query, the resolver checks to see if a more specific referral contained in the references in Referral 6 will cover Query 4. The resolver creates Referral 12 by copying the second reference in Referral 6, because the reference constrains `organization` as well as the `name` and `country` attributes that are constrained by Referral 6. The resolver uses Referral 12 in the place of Referral 6 to constrain SearchSpace further; therefore, SearchSpace = (Referral 0 $\cap$ Referral 12) = Referral 12.

The catalog referrals that cover Query 4 provide the ordered list: Candidates = {Referral 5, Referral 7, Referral 1, Referral 2, Referral 3, Referral 4}. Referral 5 indexes `name` in the United States. It is not used by the resolver, because it is more general than SearchSpace. Referral 7 will constrain the `state` attribute in the United States, so the resolver uses Referral 7 to produce Referral 13. The intersection of SearchSpace and Referral 13 gives Referral 14. Referral 14 is the final SearchSpace for Query 4, because all the other candidates are more general than Referral 14 including the unused catalog referral, Referral 1 (see Figure 5.3). The resolver uses Referral 14 to obtain responses to Query 4.

5.3.5. Example Summary

These examples show how the resolver algorithm constrains the search by using catalog functions and intersection to generate new, more specific referrals in the referral graph. The partial ordering defined by the referral graph directs query processing by identifying which catalog referrals will further constrain the search space.

Ref. ID	Source of Ref.	Template / Revised Template	Access Function (Data Repositories)
12	revised template in 6	o = "Bell Labs" and name = "Ordille" and c = "US"	Unix_10_DAF (dr130 – dr140)
13	7	state = "NJ" and c = "US"	Unix_10_DAF (dr130 – dr136) WHOIS_DAF (dr200 – dr221)
14	0 ∩ 12 ∩ 13	o = "Bell Labs" and name = "Ordille" and state = "NJ" and c = "US"	Unix_10_DAF (dr130 – dr136)

Table 5.5. Referrals generated while processing Query 4

References with revised templates provide opportunities for using the cached answers to subqueries. They also encapsulate more specific parts of the referral graph for use in answering future queries. The meta-data cache grows in specificity as queries are processed; it is tailored to the stream of queries at the resolver. In our example, fewer referrals were generated by Queries 2 and 4 than Query 1, because some of the referrals obtained or generated for Query 1 were re-used in the subsequent queries. The resolver used the meta-data cache to save the cost of contacting the distributed catalog service for each query, freeing distributed catalog and network resources for initializing other resolvers. The resolver also saved the cost of repeatedly calling the catalog functions in Referrals 2, 3, 5, and 7 for information that was saved in the meta-data cache (in Referrals 5, 6, 7, and 8). The cost of calling catalog functions includes the cost of contacting remote services as well as local processing. When remote service calls are eliminated, the system is able to support more query resolvers, because the load on the network and the remote service providers decreases.

5.4. Replica Support

Our discussion about the intersection of sets of data repositories assumes that each data repository stores a distinct partition of the name space. Performance and availability considerations often lead the administrators of naming data to replicate partitions of the name space. In this case, different data repositories store the same name space partition, and we can no longer represent the name space partition simply by the identifier of the data repository that stores it. In this section, we incorporate replicas of partitions into our intersection algorithm.

The intersection of sets of data repositories, in the absence of replicas, is linear in the number of data repositories. The elements in each set of data repositories are data repository identifiers. Data repository identifiers uniquely name data repositories. For example, an IP address uniquely identifies data repositories that are WHOIS name servers [44] while QUIPU X.500 name servers have IP or X.25 addresses [52]. Our intersection algorithm requires that the same identifier is consistently assigned to a data repository, so that identifiers can be compared for equality. It also requires the query resolver to define an ordering for its internal representation of data repository identifiers, so the identifiers can be sorted. Once these requirements are met, the sets of sorted identifiers are merged using a linear time algorithm [54], and only those data repositories occurring in both sets are retained in the intersection.

When replicas are permitted, each partition of the name space is represented, not by the identifier of one data repository, but by its *set of replicas*, i.e. the set of identifiers of the data repositories storing the partition. The locations to search for query answers is a collection of these partitions (as defined by the sets of replicas). The query resolver processes the collection by contacting one replica from each partition. Because our name service techniques support a relaxed model of consistency, it is not necessary for all referrals to have a consistent view of the set of replicas for a partition. Different referrals can contain different sets of replicas for the same partition. If two sets of replicas have a data repository in common, the resolver uses the conservative approach of considering the partitions represented by the sets to be the same. When the same partition occurs in both operands of the intersection, the resolver includes it in the result of the intersection. When the sets of replicas for the partition differ, the resolver uses the set from the referral with the most recent timestamp. A simple variant of our intersection algorithm for referrals that list unreplicated data repositories can be used to intersect referrals that list replicas, while remaining linear in the number of data repositories. The data repositories listed in each referral are sorted. The lists of sorted identifiers are merged, and only the partitions for data repositories occurring in both lists are retained in the intersection.

5.5. Consistency Support

The resolver supplies advice about the consistency of query responses by providing the collection modification time for the response. The collection modification time supports the development of interfaces that allow users to obtain more current information as needed. An example of such an interface would be one that allows users to view an answer on a terminal screen and request a more recent version of the answer. The interface then generates a request to the query resolver for a response that is more current than the collection modification time of the answer displayed to the user. A variety of other interfaces are possible, but we do not address the issues involved in selecting and creating the interfaces in this thesis.

Recall from Chapter 3 that the collection modification time is the timestamp of the oldest active catalog information used in answering the query. Tuples added after the collection modification time might not be reflected in the response. This section describes how the resolver calculates the collection modification time from timestamps provided by the active catalog. Referrals and the collection of tuples returned by data access functions have timestamps. The timestamp represents the time after which information in the referral or collection of tuples may have changed.

The resolver calculates the collection modification time from the timestamp for the SearchSpace referral and the timestamps for the collections of tuples returned by data access functions. The distributed catalog service maintains the current general referrals for each relation. It supplies its current timestamp with the referrals it returns to the query resolver. When a new referral is generated from a catalog referral, its timestamp is the earliest of the timestamps of the components (the catalog referral and referrals returned by the catalog functions) used to generate it. The resolver calculates the timestamp of SearchSpace as it generates the referral; the timestamp is the earliest timestamp of the referrals that are intersected to form SearchSpace.

The resolver initializes the collection modification time to the timestamp of the value of SearchSpace used to answer the query. When the resolver obtains responses for the query, the data access function or the data cache provide a timestamp for the collections of tuples they supply to the response. If the timestamp is earlier than the collection modification time, it becomes the new value of the collection modification time. The collection modification time is returned to the user with the query response, and stored with the response in the data cache.

Users can request a more current response to a query by specifying a lower bound on the collection modification time. The distributed catalog service and the access functions satisfy a request for answers with timestamps after the lower bound whenever possible. Since the distributed catalog service, data repositories, and remote catalog functions may be unavailable from the network, the resolver can continue to answer some queries, but not queries for more current information, by successfully obtaining new information before removing older entries from its caches. By supporting an implicit lower bound on the collection modification time, the resolver can automatically enforce a timeout invalidation algorithm similar to the one used by the Domain Name System. The implicit lower bound is set to a known interval before the current time. Removing cache entries with timestamps before the implicit lower bound then limits the inaccuracies introduced into query answers by old information in the resolver cache.

Chapter 6

MAINTAINING THE ACTIVE CATALOG

The two primary questions in maintaining the active catalog are how to add new data repositories and how to add catalog functions to the referral graphs generated by query resolvers. All the referrals generated by resolvers originate in the set of referrals provided by the distributed catalog service. The resolvers generate referrals by following referral graph paths to more specific referrals. Thus, to be accessible to query resolvers, new data repositories and catalog functions must be added to referral graph paths that originate in the distributed catalog service.

Section 6.1 describes the path reachability requirements that must be satisfied as data repositories and catalog functions are added to the active catalog. Section 6.2 explains how to satisfy these requirements when adding data repositories to the active catalog. Section 6.3 explains how to satisfy path reachability requirements when adding catalog functions to the active catalog. The goal of this chapter is to show the plausibility of building and maintaining referral graphs. A full treatment of active catalog maintenance would require formal definitions and a proof that the maintenance algorithms preserve referral graph invariants.

6.1. Referral Graph Requirements

As described in the examples in Chapter 5, the distributed catalog service supplies the most general referrals for a relation. The most general referrals are the base referral and referrals that index a single attribute on a relation. The base referral covers every query on a relation. By definition, the base referral is a data referral, so catalog referrals must be generated by other referrals in the distributed catalog service. A referral with one wildcard attribute in its template is the most general catalog referral for that attribute. It is more general than any other referrals that include the attribute, and can generate other catalog referrals. From the base referral and the most general referrals for each attribute, all other referrals in the referral graph are generated.

The most general referrals and all referrals that can be generated from them form the *global referral graph* for a relation. The two primary questions in maintaining the active catalog are how to add new data repositories and catalog functions to the global referral graph. The global referral graph does not actually exist in the distributed catalog service, but parts of it are created as needed by resolvers. A resolver establishes the search space for a query by walking the global referral graph to the most specific referral in the graph that covers the query. The search space is the intersection of the data referrals that cover the query. When a resolver processes a catalog referral, it produces a referral that is more specific than the catalog referral (see Section 5.2). When a resolver creates a new referral by intersecting the search space and a data referral, it produces a new search space referral identical to or more specific than the operands of the intersection (see Section 5.3.1). To be used by a query resolver, new data repositories and catalog functions must be reachable by walking the global referral graph.

The reachability of new data repositories and catalog functions must obey the consistency requirements of the active catalog. A referral must represent all changes to the data repositories through the timestamp of the referral, and may represent subsequent changes as well (see Section 5.5). The following definition describes the requirements for the reachability of a data repository in the global referral graph.

> *Data Repository Reachability:* A data repository is reachable in a referral graph if a query resolver generates a referral to the data repository for every query that has a non-empty answer at the data repository at the timestamp of the referral.

Catalog function reachability differs from data repository reachability, because the query resolver only needs the catalog function to be reachable when it can improve the performance of a query. A catalog function can improve performance if it adds additional constraints (i.e. attribute values) to the search space for a query. The following definition describes the requirements for reachability of a catalog function in the global referral graph.

> *Catalog Function Reachability:* A catalog function is reachable in a referral graph if a query resolver can generate a referral to the catalog function.

A query resolver may not generate a referral to a reachable catalog function if another path through the global referral graph constrains the attribute(s) in the template of the catalog function. For example, if the referral graph contains referrals to catalog functions with templates: `(state = "*" and dept = "CS")` and `(state = "WI" and dept = "*")`, a query resolver will only generate the first referral encountered during the graph traversal. The catalog function in the first referral encountered will constrain both the `state` and the `dept` attributes, so the second referral would not add constraints to the search space. The addition of a catalog function to the global referral graph must satisfy two requirements. First, the catalog function must be reachable. Second, the catalog function must preserve data repository reachability for all data repositories in the name space.

6.2. Adding Data Repositories to the Global Referral Graph

Adding a new data repository to the referral graph requires information about the contents of the data repository. Knowing the format and protocol of the data repository allows us to identify or construct a data access function for the repository. Any logical constraints on the values of attributes in the data repository determines where *not* to place the data repository in the referral graph. For example, if all tuples in the data repository have the same value for the `country` attribute, then the data repository need not be added to the parts of the referral graph that constrain searches in other countries. The *logical search space* for a data repository is a conjunctive predicate, `p`, on the attributes of the data repository such that all tuples in the data repository are guaranteed to satisfy `p`. A data repository only needs to be added to a referral if its logical search space is consistent with the logical search space (template) of the referral. Describing the logical search space of a data repository with predicates that contain other logical operations (e.g. disjunction or negation) or comparisons (e.g. inequality comparisons), or characteristics other than attribute values (e.g. semantic relationships between knowledge in the data repository and other kinds of knowledge) is an area of future research.

There are two techniques for incorporating data repositories (or catalog functions) into the active catalog. One technique adds the data repository to a referral in the distributed catalog service. The other technique adds the data repository to an existing catalog function. Figure 6.1 provides an example of incorporating a data repository into referrals. `Dr55` is added to Referrals 1 and 2, because its logical search space is consistent with the templates of the referrals. The data repository is not added to Referral 3, because the value for the `organization` attribute in its logical search space is inconsistent with the template of Referral 3. When Referral 1 is used, both `dr55` and `dr76` are contacted for data responses. When Referral 2 is used, `dr55` is contacted for data responses and the catalog function `cf` is contacted for additional data repositories to search. Instead of incorporating `dr55` into Referral 2, it could be added to catalog function `cf`. Adding a data repository to a catalog function is discussed further in the remainder of this section.

There are two techniques for maintaining consistency while adding a data repository to the global referral graph. First, *bottom-up addition* occurs when the new data repository belongs to an existing name service, like

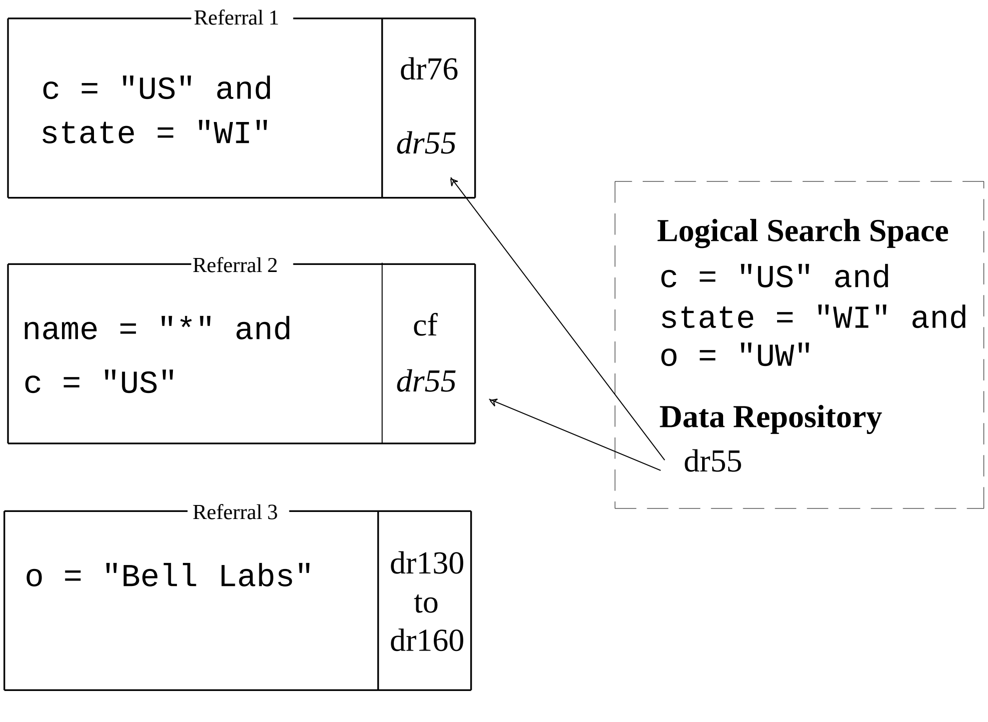


Figure 6.1. Adding a data repository to referrals with consistent templates

(The logical search space for data repository `dr55` *is consistent with the templates in Referrals 1 and 2, but inconsistent with Referral 3. The data repository is added to the referrals with consistent templates.)*

X.500, that is already incorporated into the active catalog. A set of catalog functions already generates referrals to data repositories in the existing name service. The data repository is added to leaves of the referral graph, because it is automatically incorporated into those catalog functions. Second, *top-down addition* occurs when the data repository does not belong to a service that has previously been incorporated into the active catalog. Since the data repository does not belong to an existing service, it will not be found automatically by existing catalog functions. The referral graph is traversed to find the referrals or catalog functions that need to incorporate the data repository.

Bottom-up addition satisfies the data reachability requirement if two conditions are met. First, the catalog functions for an existing name service must correctly incorporate new data repositories into their indices. Second, the catalog functions must hold positions in the global referral graph that allow them to include the new data repositories in relevant searches. For example, a catalog function that indexes `state` for the X.500 name service can only be included in Referral 2 in Figure 6.2a if a new data repository will never be added for a country other than the United States. If data repositories for different countries can be added to X.500, then the catalog function must be included in Referral 1 to allow for possible new values for `country` in new data repositories. A simple way to satisfy the second condition is to provide a catalog function that covers the existing service for each attribute in the relation. The most general referral for an attribute incorporates the catalog function that indexes that attribute for the existing service. Because the catalog functions for the existing name service hold positions at the highest levels of the referral graph, their results are always incorporated into the search space for queries.

The goal in top-down addition is to trim the data repository from as many query search spaces as possible while still satisfying the data repository reachability requirement. For example, `dr56` in Figure 6.2b is in the global logical search space; its logical search space is unconstrained and consistent with every referral in Figure 6.2a. Adding data repository `dr56` to Referral 1 in Figure 6.2a ensures that every query covered only by the referral graph path rooted at Referral 1 will have `dr56` in its search space. If the logical search space for `dr56` was `(c = "US")`, we could add `dr56` to the more specific referral, Referral 2, and not Referral 1. This would limit the search spaces that include `dr56` to search spaces for the United States. When a data repository is in a constrained logical search space, we can avoid adding the data repository to some referrals and, therefore, to the search spaces for some queries.

The top-down addition algorithm satisfies the data repository reachability requirement by adding the data repository in logical search space `p` to each referral graph path that is consistent with `p`. It always adds the data repository to the base referral. It then treats `p` as a query, and considers whether each of the most general catalog referrals in the distributed catalog service covers `p`. First, if the referral covers `p`, the algorithm uses the referral to process `p` as if it was a query. This processing generates a path through the referral graph; the path contains referrals that cover `p`. The most specific catalog referral in the path is the referral, R, where the data repository must be added. The data repository is added to a catalog function in R or to R itself. If a catalog function incorporates the new data repository, the repository and its data access function are added to all the referrals that are created by the catalog function and are also consistent with `p`. If a catalog function does not support the addition of new data

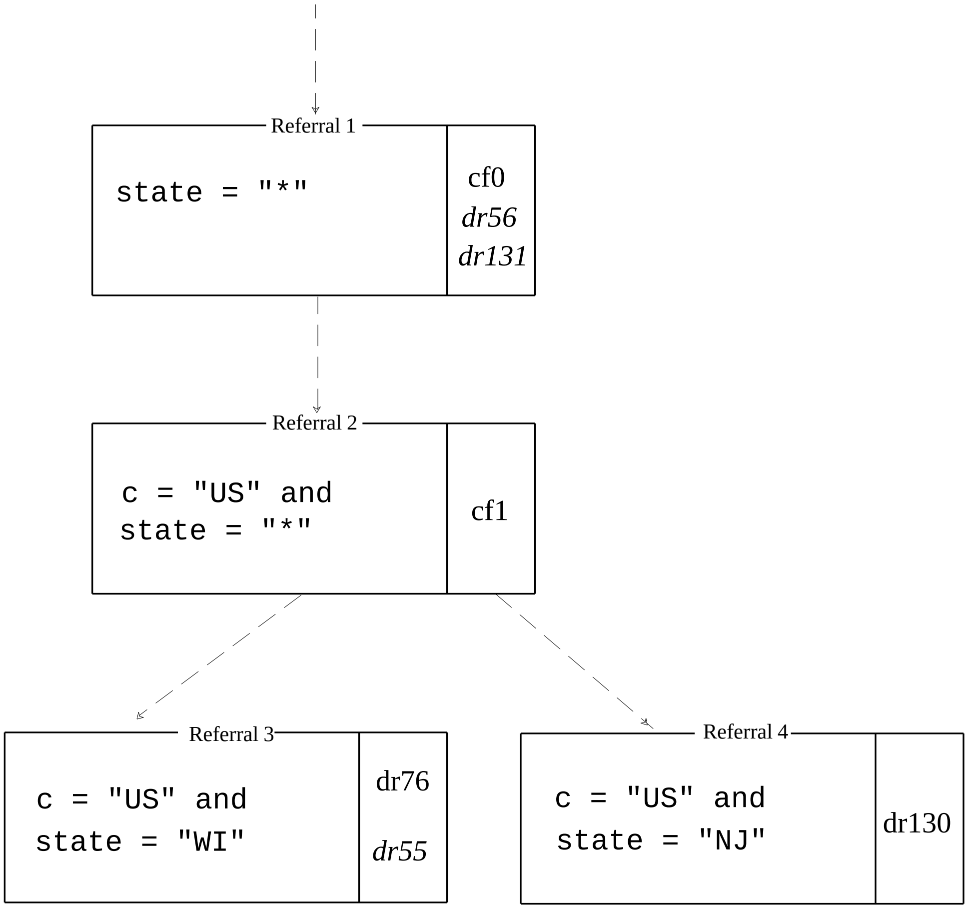


Figure 6.2a. Part of the global referral graph for the `People` relation

Logical Search Space (p)	Data Repository To Add
c= "US" and state = "WI" and o = "UW"	dr55
	dr56
o = "Bell Labs"	dr131

Figure 6.2b. Data repositories added by top-down addition to the graph in Figure 6.2a

(The top-down addition algorithm adds each data repository in Figure 6.2b to the global referral graph in Figure 6.2a as shown by the italic data repository identifiers in the graph.)

repositories, the data repository can be added directly to R or to a referral that is more general than R. Since the most general referral in the path is controlled by the distributed catalog service, the data repository can always be added to one referral in the path. More general referrals need not incorporate the data repository if their catalog functions use the more specific referral that incorporates the new data repository when constraining queries in `p`. For example, in Figure 6.2, the last catalog referral generated in the path from Referral 1 that covers `p` for `dr55` is Referral 2. The data repository is added to all referrals generated by catalog function `cf1` that are consistent with `p`. In particular, `dr55` is added to Referral 3, but not Referral 4. Referral 1 does not need to incorporate `dr55` directly, because the catalog function in Referral 1 uses Referral 2 to constrain queries for `c = "US"`.

Second, if the data repository was not added to the path from the most general referral in the previous step, the top-down algorithm adds the data repository to the most general referral or to a catalog function in that referral. Since each of the most general catalog referrals has one wildcard attribute, these referrals are logically consistent with every `p`. For example, `dr56` and `dr131` are added to Referral 1, because they are not covered by Referral 1 but are logically consistent with it. The top-down addition algorithm satisfies the data repository reachability requirement for the global referral graph, because each data repository is added to all the paths through the referral graph except those that are inconsistent with `p`. If `p` is inconsistent with a path, then no tuples that satisfy queries covered by the path exist at the data repository.

Automating the top-down algorithm requires that catalog functions accept requests to add data repositories to their indices. The catalog functions can simply include the new data repository in all referrals that are consistent with the predicate of the data repository. More sophisticated catalog functions will gather indexing information for the new data repository. When catalog functions do not support the addition of data repositories to their indices, the data repositories can be incorporated into a more general referral in the referral graph path or finally in the most general referral in the path. Adding new data repositories to existing catalog functions assumes that the catalog functions are trustworthy, i.e., that the catalog functions will return referrals that correctly reflect the contents of the data repository. Handling catalog functions that are untrustworthy is an area of future research.

6.3. Adding Catalog Functions to the Global Referral Graph

When a catalog function is added to the global referral graph, it must satisfy the catalog function reachability requirement. We can satisfy this requirement by adding the catalog function to all referral graph paths consistent with the template of the catalog function. This technique is inefficient, because the catalog function is added to paths that the query resolver need not follow in constraining queries covered by the catalog function. Only queries that have all of the attributes in the template of the catalog function are covered by the catalog function, so we can limit the places where the catalog function is added to some of the paths that constrain the attributes in the template.

As a general organizational principle, we always add a new catalog function to the referral graph paths that have the same wildcard attributes as the template for the catalog function. This organizational principle provides initial paths through the referral graph that include the catalog function. Adding a catalog function to the paths with the same wildcard attributes is not sufficient to guarantee catalog function reachability. The catalog function becomes reachable along these paths, but the other constraints in its template may not be found when needed to limit the search space for a query. For example, consider two catalog functions that have the templates: `(state = "*" and dept = "CS")` and `(city = "Madison" and state = "WI" and dept = "*")` respectively. A query resolver finds and uses the first catalog function to constrain the search for a query covered by both templates. The query resolver will not find and use the second catalog function, because the wildcard attribute for the second function is already constrained in the search space for the query. Catalog function reachability is violated, because the second catalog function is not found when it can constrain the `city` attribute for the query. To satisfy catalog function reachability, the catalog function in this example must be added to another path in the global referral graph, specifically the path from the most general referral for `city`. When adding a catalog function, we must determine if the most general referral, `G`, for each non-wildcard attribute in the catalog function template generates a referral for the value of the attribute. If `G` does not generate a referral, the catalog function must be added to a path that begins at `G`.

Similarly to data repositories, there are two techniques for maintaining consistency while adding a catalog function to the global referral graph. First, bottom-up addition is used to maintain catalog functions for an existing name service. Bottom-up addition automatically adds catalog functions to referrals generated by a more general catalog function for an existing name service. The new catalog function is incorporated into the general catalog function by the general function's maintenance procedures. For example, a catalog function that indexes `state` for a new country in the X.500 tree is automatically incorporated by procedures that maintain the `state` indices in X.500. Bottom-up addition also occurs when a catalog function is replaced by multiple catalog functions. For example, a single catalog function that indexed all state and province names in the world was previously replaced by the catalog function subgraph beginning with Referral 1 in Figure 6.2. Replacing a catalog function with multiple catalog functions can improve performance by dividing processing load across multiple remote catalog functions or decreasing the size of a local catalog function. When bottom-up addition adds catalog functions with non-wildcard values in their templates, it must ensure that non-wildcard attribute values are also constrained by the referral graph paths for those attributes.

Second, top-down addition of catalog function occurs by traversing global referral graph paths that have wildcard values for the same attributes. The top-down algorithm creates a query that exactly matches the template of the catalog function by replacing each wildcard in the template with a randomly generated value. The query is processed using a general referral that has wildcards for one or more of the attributes with wildcards in the template of the new catalog function. This processing generates a path through the referral graph; the path contains referrals that cover the query. The most specific catalog referral on the path that covers the original template (with wildcards) is the referral, R, where the new catalog function must be added. If the template of the new catalog function is more specific than the template of R, the new catalog function is added to a catalog function in R or to R itself. If a catalog function incorporates the new function, it includes the new catalog function in referrals that are consistent with the new function's template. For example, this algorithm would add a catalog function for the template `(c = "CA" and state = "*")` to `cf0` in Figure 6.2. If a catalog function does not support the addition of a new more specific catalog function, the new catalog function and its template can be added as a catalog function and revised template respectively to R or to a referral that is more general than R. Automating the top-down addition of catalog functions requires that catalog functions accept requests to add a new catalog referral to their scope. Catalog functions can keep of list of more specific referrals to return when appropriate. When none of the more specific referrals covers the query, the catalog functions use their original algorithm to generate a referral.

When a catalog function is added to the global referral graph, it must also satisfy data repository reachability for all data repositories in the name space. The catalog function must correctly index those data repositories in the logical scope of its template. Moreover, all data repositories in the logical scope of the template must be indexed. When a catalog function is added for an organization by the naming authority of that organization, the naming authority is responsible for ensuring that reachability is satisfied for data repositories in the logical scope of the catalog function. When the top-down algorithm adds a catalog function, it can check for other catalog functions with templates that are identical to the template of the new catalog function. If the templates are identical, then the top-down algorithm must determine whether to add the new catalog function to the references for the referral, or to replace the old catalog function with the new catalog function. For example, a new catalog function for the template `(c = "US" and state = "*")` conflicts with `cf1` in Referral 2. In the absence of information that one catalog function supercedes another, the conservative solution of including both catalog functions in the referral preserves data repository reachability.

Chapter 7

PERFORMANCE RESULTS

Three issues are important in evaluating the performance of our query processing framework. First, we must determine whether an active catalog can constrain the search space for queries in a real environment. Are there attribute values that will isolate queries to a few data repositories in the global name space? Are users likely to know those attributes? Second, we must determine the performance advantages of active cataloging and meta-data caching given the existence of constrained search spaces. Can a user find information in the global name space quickly? Third, we must analyze the scaling behavior of our query processing framework for workloads of multiple users. How well will our query processing scale to millions of users and tens of thousands of data repositories?

In this chapter, we address these issues by presenting the results from three performance studies. The first study uses our prototype descriptive name service, Nomenclator, to examine the performance of an active catalog in a real environment, the X.500 name space [22]. The second study uses Nomenclator to examine the performance of an active catalog and meta-data caching for queries from a single user. The queries are constrained by Nomenclator to 0 through 100 percent of the data repositories in a search space of 1000 data repositories. Finally, the third study uses an analytical performance model of Nomenclator to examine the performance of workloads where millions of users search tens of thousands of data repositories.

Our current studies do not measure the effects of processing a single query in parallel. On the Internet, the performance of parallel data repository access is limited by the speed of sending requests and receiving responses at a query resolver. Even when multicast operations are available, the speed of processing responses at a query resolver limits parallelism. Our query processing framework should benefit when parallel access to data repositories is used, because it continues to reduce the load on data repositories and the network by constraining the search space. Parallelism potentially allows more queries (with larger search spaces) to be answered within a reasonable response time. Additional research is necessary to address the limits of parallel query processing in a large internet environment.

We do not analyze the costs of maintaining the indexing data used by catalog functions. These costs vary from zero for catalog functions that use static indexing information to the costs of periodically updating a dynamic index. In any case, the costs of maintaining the indexing data for catalog functions should have a minimal impact on performance, because the frequency of read requests far exceeds the frequency of update requests and updates can be processed when network and server load is low. The costs of dynamically loading access functions are also not analyzed in our studies, because dynamic loading costs are typically low [19], and access functions change very infrequently. Most catalog functions will come from a standard set of access functions that will be varied by the data used to initialize them.

Section 7.1 presents the results of our X.500 experiments, and Section 7.2 presents the results of our single user experiments. These experiments show how Nomenclator improves the performance of single queries and reduces the impact of the individual user on the query processing environment. Section 7.3 presents our analytical modeling results. Our analytical model shows how Nomenclator manages queries from many users to achieve scalable system performance. Section 7.4 summarizes our performance results.

7.1. X.500 Experimental Results

The experiments described in this section 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 and meta-data caching in improving performance. In our experiments, catalog functions and meta-data caching consistently improve the performance of the X.500 `SEARCH` command.

These initial experiments do not examine the cost of communicating with a distributed catalog server or the effects of meta-data caching on network and server load. We address these concerns in Section 7.2. These experiments also do not examine the performance of multi-user workloads. We address these concerns in Section 7.3. 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 7.1.1 provides an overview of X.500. Section 7.1.2 describes the test environment, and Section 7.1.3 presents our results.

7.1.1. X.500 Overview

X.500 [22] standardizes OSI directory services for locating people and application objects. QUIPU [52], currently the most popular X.500 implementation, is used in a pilot name space including over 350 organizations in 13 countries [86]. 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 7.1, 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 7.1, 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 [51].

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 7.1 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 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

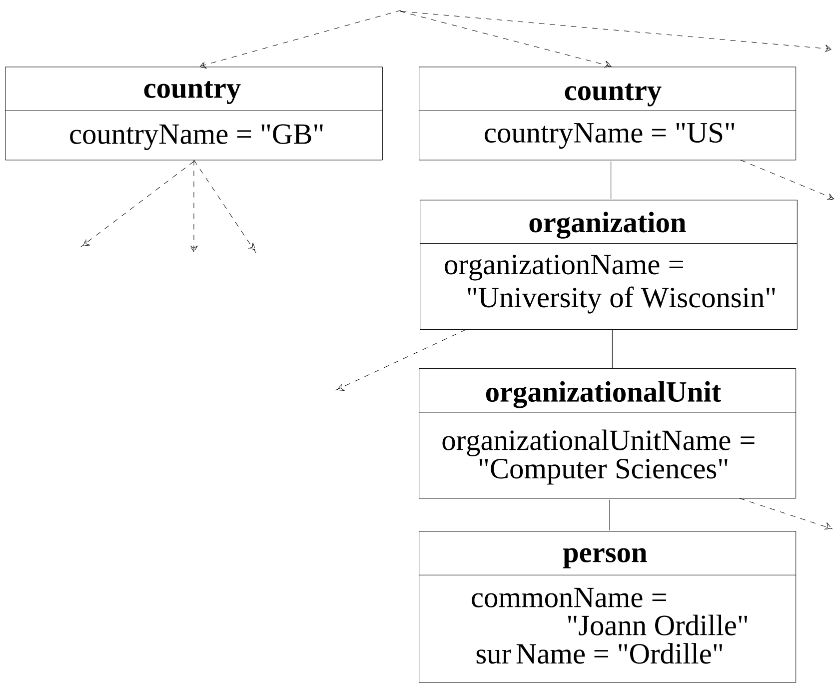


Figure 7.1. Sample X.500 directory information tree

cost of the operation. Users must navigate the DIT to search the organization subtrees of a country. We address these problems further in Section 7.1.2.

The DIT is partitioned along the arcs of the tree similarly to the Domain Name System [67]. 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 meta-data and data caching techniques for improving the performance of `SEARCH` and other descriptive queries.

7.1.2. Environment

Our experiments were performed on the United States portion of the QUIPU pilot X.500 name space [87] 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 [86].

Our experiments use a prototype implementation of Nomenclator that processes queries like those in Table 7.1. It includes a query resolver, but no distributed catalog service. The referrals that would typically be returned by the distributed catalog service are preloaded into the meta-data cache, and the access function 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 queries like those in Table 7.1, and returns results in the same format as the Nomenclator prototype. 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 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 sequentially process queries to different data repositories 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.

The Nomenclator prototype includes a catalog function that constrains the `state` attribute in the United States, called `US_States`, and a catalog function that constrains the `name` attribute in the United States, called `US_Names`. 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 US_Names catalog function uses an optional organization attribute value, as well as name, to distinguish between data repositories. US_Names builds referrals by combining the results of calls to more specific Org_Names catalog functions for each organization in the United States. 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 uses a bit vector filter (see Section 3.2) 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.

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, that lists the permitted classes for children of an object in QUIPU, provided inadequate information for this task. Naming administrators typically took a liberal approach to the attribute by listing all the logical possibilities for subtrees. 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.

7.1.3. 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 7.1. 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 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.

Table 7.2 gives the results of our experiments. The number of items returned by each query is listed. The performance measurements are listed for each program along with the number of servers they contacted in processing the query. The 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 results in the "Improvement Factor" column. Dividing the X.500 performance by the improvement factor gives the Nomenclator performance.

The improvement factors reflect the benefits accrued in Nomenclator from trimming the search space. Note that the queries with the greatest improvement in performance are those where the number of servers contacted is most reduced. The lower improvement in Query 2 reflects the predominance of data repository processing costs in

No.	Query
1	`select * from People where name = "Ordille" and c = "US"`
2	`select * from People where name = "Miller" and c = "US"`
3	`select * from People where name = "Miller" and state = "WI" and c = "US"`
4	`select * from People where name = "Ordille" and state = "WI" and c = "US"`
5	`select * from People where name = "Miller" and state = "HI" and c = "US"`
6	`select * from People where name = "Ordille" and state = "HI" and c = "US"`
7	`select * from People where name = "Miller" and o = "University of Wisconsin" and c = "US"`
8	`select * from People where name = "Chaillou" and o = "University of Wisconsin" and c = "US"`

Table 7.1. Test queries and their identifying numbers

Query	Items	X.500 Time	X.500 Servers Contacted	Nomenclator Time	Nomenclator Servers Contacted	Improvement Factor
1	1	221.4	34	13.5	1	16.4
2	160	1563.8	34	1468.5	11	1.1
3	2	324.0	34	15.9	1	20.4
4	1	309.9	34	13.4	1	23.1
5	0	351.0	34	9.1	0	38.6
6	0	218.7	34	9.5	0	23.0
7	2	16.1	1	15.4	1	1.1
8	0	10.5	1	7.0	0	1.5

Table 7.2. Performance of X.500 and Nomenclator for the test queries

(Times are reported in seconds. X.500 and Nomenclator performance for cold data caches.)

executing this query. It illustrates that the cost of contacting data repositories with no data relevant to the query becomes insignificant as the ratio of data repository processing time for large answers to processing time for null answers rises. The experiments described in Section 7.2 use an optimized data repository to show that substantial performance improvements can be achieved when large amounts of data are processed by data repositories. As data repository processing is optimized, our query processing techniques have greater impact.

Our results incorporate the benefits of meta-data caching, because initial referrals and catalog functions are loaded into the query resolver before the experiments begin. In Section 7.2, we examine the costs of initializing catalog functions and retrieving referrals for the meta-data cache.

7.1.4. 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.

7.2. Single User Experimental Results

In our second study, we evaluate the benefits and costs of using our techniques when queries are constrained to 0 through 100 percent of the data repositories in a relation. Our goal is to identify whether performance will be acceptable in Nomenclator's intended operating range where queries are isolated to some small percentage (30 percent or less) of the data repositories in the search space. We know from our previous study that we can isolate some queries to this percentage of the data repositories. We are also interested in verifying that there are no bottlenecks to single query performance in our system. Our experiments compare the performance of the naive algorithm that searches everywhere with our query processing and meta-data caching techniques. The naive algorithm is now used in several name services, including X.500 and meta-services that query other name services like the Knowbot Information Service [34].

7.2.1. Environment

During the experiments, Nomenclator's distributed catalog server, the query resolver, the naive algorithm, and the data repositories all executed on different DECstations running Ultrix in a local area network. We chose to use a local area network for our tests, because we have more control over this environment than over the wide-area network. We were able to ensure that other network and computing activities did not interfere with our experiments. Experiments in the local environment are conservative, because wide-area networks have greater delays that make active cataloging and caching results look even better.

To attain the scale of our intended wide-area application, we created a program that implements a variable number of data repositories on one host. The program answers a query differently depending on the data repository address presented with the query. We ran the program on 10 DECstations; each DECstation supported 100 data repositories during the experiments. Using one program per host and only processing sequential queries prevented any context switching or query processing conflicts between data repositories on the same host.

We tested against a relation stored on 1000 data repositories. The relation had two attributes. One attribute in the relation contained one byte values that occurred in 0, 25, 50, 75 or 100 percent of the data repositories in the relation. This attribute was specified in the selection predicate of the query. The other attribute contained 1 or 1000 byte value depending on the test. This attribute value was returned in the query response. The experiments occurred during non-peak weekend or evening hours on otherwise idle workstations.

Nomenclator used one catalog and one data access function during the experiments. An initial referral to the catalog function was available from the distributed catalog service. After being started by Nomenclator, the catalog function used an internal Nomenclator relation to retrieve bit vector filters that described the hash values of the attribute to be selected at each data repository. The catalog function compared the hash value of the attribute in the query with those in the filter to decide which data repositories to include in the referrals it generated for the query resolver. Data caching was disabled during the experiments, so we can evaluate the performance of meta-data caching in isolation.

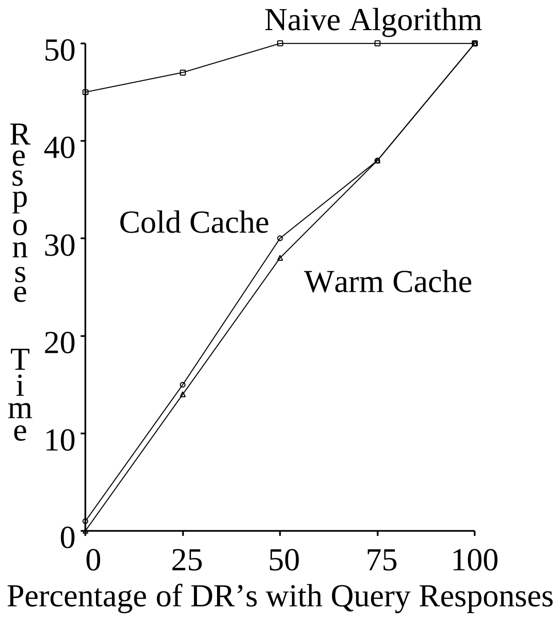


Figure 7.2. Response time results for 1 byte response tuples

(Response time in seconds for each data distribution pattern.)

7.2.2. Results

Our experiments measured the performance of queries that selected 0 to 100 percent of the data repositories. Each query was run by the naive algorithm, by Nomenclator with a cold meta-data cache, and by Nomenclator with a warm meta-data cache. When Nomenclator had a cold meta-data cache, it initialized its cache from the distributed catalog service, called the catalog function, and then contacted the data repositories for query responses. The warm cache results report the performance of the second and subsequent queries in a series of identical queries. Nomenclator finds the cached result of the catalog function call and does not re-call the catalog function.

We measured the response time of each query. We also measured the total number of bytes transferred by all network messages during query processing. The total bytes transferred is a metric for the load placed on the underlying system and the computer network. The measurements reported here are the average of several runs for a query.

Figure 7.2 reports the response time measurements for the queries where the data repositories returned one byte tuple responses, and Figure 7.3 reports the number of bytes transferred by those queries. The number of bytes

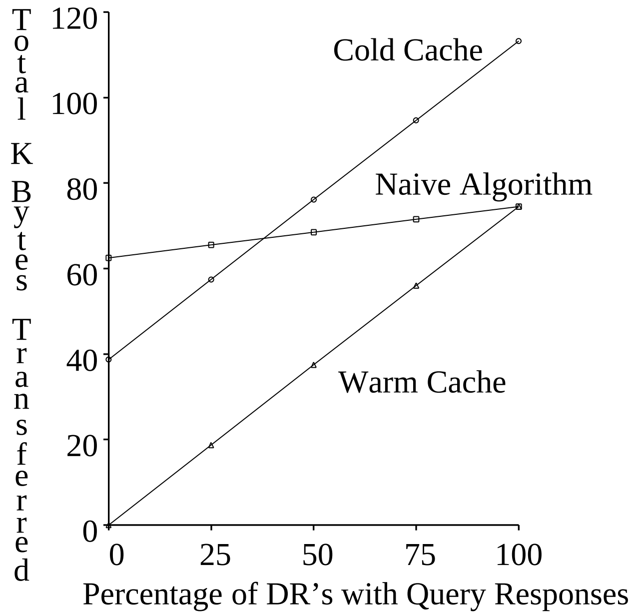


Figure 7.3. System Load Results for 1 Byte Response Tuples

(Thousands of bytes transferred for each data distribution pattern. Byte count includes queries sent, responses received, meta-data initialization, and communications protocol overhead.)

transferred is significantly larger than the 1000 bytes of tuple responses, because it includes the cost of sending the query to the data repository and the protocol overhead for packaging the query and the response. In the case of the cold cache, it also includes the size of messages used to retrieve referrals and initialize the catalog function. Figure 7.4 reports the response time measurements for queries where the data repositories returned 1000 byte responses. The x-axis of each graph indicates the percentage of data repositories containing query answers.

7.2.3. Discussion

Our experiments show that our techniques to eliminate data repositories from the search space can dramatically improve response time. As we anticipated, Figures 7.2 and 7.4 report a linear relationship between the number of data repositories contacted and the response time. Our techniques successfully eliminate unnecessary work from query processing without introducing new bottlenecks. Both graphs show significant response time improvements, because communications latency is an important performance constraint that is reduced by our query processing techniques. Our measurements were taken on a local area network under optimal conditions; wide-area network improvements are even greater due to the increased latencies in those networks. As networks become large, latency worsens faster than bandwidth and must be addressed by optimization techniques like ours.

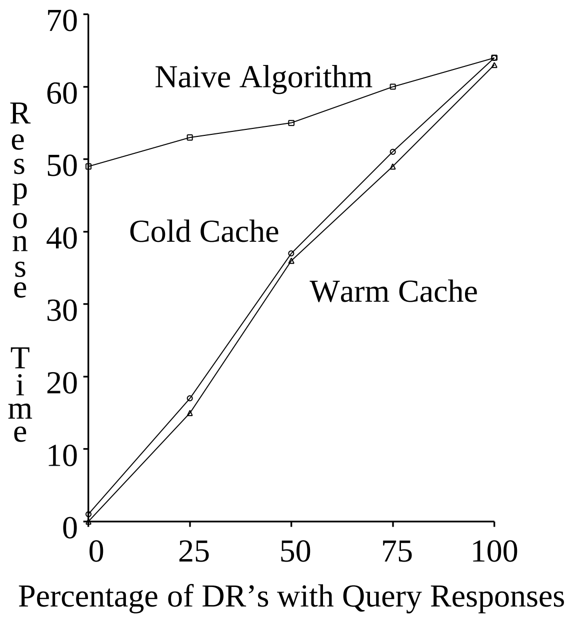


Figure 7.4. Response time results for 1000 byte response tuples

(Response time in seconds for each data distribution pattern.)

Figure 7.4 shows that response time savings are significant even when a large amount of data is returned. Since we expect typical name service queries to return a few thousand bytes, Figure 7.4 shows that even large name service queries will be answered quickly. When 30 percent or less of the data repositories contain responses, both Figures 7.2 and 7.4 report a 70 percent or more increase in performance. Queries that previously searched the global name space for minutes (or remained unasked because they were too costly) can now be answered in seconds.

Our experiments show a favorable tradeoff between the system load incurred by Nomenclator during query processing and the system load it eliminates by constraining the search space. Figure 7.3 shows that meta-data caching keeps the system load, as indicated by number of bytes transferred, below the load of the naive algorithm. Since obtaining referrals from the distributed catalog and initializing catalog functions has a data transfer cost, Nomenclator exceeds the load of the naive algorithm when more than 35 percent of data repositories are contacted. In our operating range of 30 percent or fewer data repositories contacted, the active catalog consistently reduces system load over the naive algorithm. The benefits in bandwidth of eliminating unnecessary queries to data repositories outweighs the cost of retrieving meta-data, and meta-data caching eliminates even this cost. System load is also decreased, because we substitute an interaction with the distributed catalog service for hundreds of interactions with data repositories. Even when Nomenclator exceeds the bytes transferred by the naive algorithm, the elimination of hundreds of interactions achieves significant improvements in response time. This improvement exists, because latency reduction is critical to large-scale name service query optimization.

Meta-data caching also leads to improved performance in multi-user workloads. As Figure 7.3 shows, meta-data caching can reduce the data transferred in retrieving referrals and initializing catalog functions. This reduction in load at the distributed catalog server eliminates bottlenecks in multi-user workloads and increases the ability of our system to scale to many users. By protecting data repositories from unnecessary queries, catalog functions also increase the ability of our system to scale to many users. Figures 7.2 and 7.4 show less dramatic performance gains from meta-data caching for single users, because latency was low in our test environment. This improvement will be larger for the greater latencies of the wide area environment. In addition, when small numbers of data repositories are contacted, improvements of a few seconds in response time from meta-data caching can be quite significant, because they often constitute the majority of the processing time in this operating range.

7.3. Multi-User Analytical Model Results

In this section, we present a performance model for our prototype descriptive name system, Nomenclator. The model explains the results of the single user experiments described in Section 7.2. It also predicts the performance of our system for large-scale workloads of millions of users searching tens of thousands of data repositories. The model is an open, separable queueing network that we solve with well-known queueing theory techniques [57]. In Section 7.3.1, we present a model for the service demands in Nomenclator. In Section 7.3.2, we present measurements of input parameters of the model for a local area network. Using these input parameters, we validate our model of service demands. In Section 7.3.3, we solve our queueing network model to obtain average response times for multi-user queries. We also present a formula for calculating the minimum number of instances of a system component needed to support different query arrival rates. In Section 7.3.4, we analyze the results of our model for a large internet environment. The model results show that meta-data caching promotes scaling by reducing the number of distributed catalog servers to a small percentage (e.g. 0.004%) of the number of users. It also shows that little data repository replication is required in our target environment, and that the number of required query resolvers is satisfied naturally if each department or small organizaton supports its own resolver.

7.3.1. Model

Nomenclator has three components: (1) a query resolver, (2) a distributed catalog server, and (3) a set of distinct data repositories. Each of these components can be replicated to improve performance (see Section 7.3.3).

Figure 7.5 shows a queueing network model for Nomenclator. Queries arrive at a query resolver, *QR*. If the

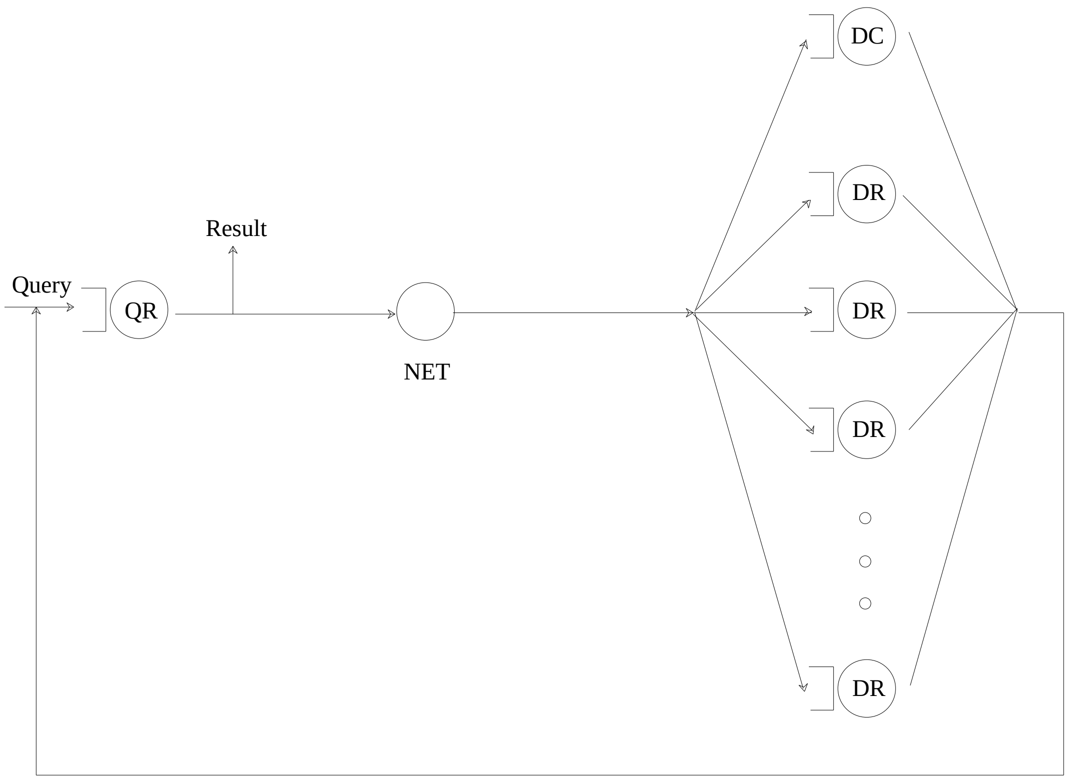


Figure 7.5. Queueing network model for Nomenclator

(The model includes a query resolver queueing center, QR, a network delay center, NET, a distributed catalog queueing center, DC, and several data repository queueing centers, DR.)

meta-data cache misses, the query resolver contacts a distributed catalog server, *DC*, for meta-data to constrain the search space. After the query resolver constrains the search space, the query resolver contacts the subset of relevant data repositories, *DR*, for answers to the query. Communication with the distributed catalog service and data repositories is delayed by the network. We model *QR*, *DC* and *DR* as queueing service centers because they service one query at a time. We model the network, *NET*, as a delay service center, because several queries can traverse the network in parallel. For example, while one query traverses the network to request referrals from the distributed catalog server, another query can traverse the network to retrieve a response from a data repository.

The model allows us to study the contributions of meta-data caching and active cataloging to query performance. The model addresses the performance issues that arise when the data cache misses, so it does not include a model of data cache operations. A data cache hit improves performance by eliminating searches of the network and reducing the query load on data repositories.

Nomenclator achieves parallelism in query processing by executing several different queries in parallel, but the query resolver contacts the data repositories for any particular query sequentially. Both the sequential execution of one query and the parallel execution of multiple queries are reflected in the model. Each query is processed sequentially, because it visits one service center at a time. Multiple queries can be processed in parallel, because a different query can be processed at each queueing center and multiple queries can traverse the network simultaneously.

Table 7.3 describes the input parameters to the model. All model inputs and outputs are averages. We assume that the generation of naming requests is a Poisson process with arrival rate λ. A Poisson arrival rate is justified, because our system is large and the Poisson assumption has worked well for other models of large naming systems [16]. S_{DC} and S_{DR} are the service times for individual requests to a distributed catalog server and data repository respectively. The service time for the network, S_{NET}, includes the round-trip delay of establishing transport connections when applicable, and the round-trip delay between requests and responses. We simplify our model by assuming that the network has sufficient capacity to service our requests. This assumption is reasonable, because gigabit networks far exceed the bandwidth requirements of our system. There are three classes of service time specific to the query resolver: S_{QRl}, S_{QRc} and S_{QRd}. S_{QRl} is the mean time to search the meta-data cache for referrals. S_{QRc} is the mean time the query resolver spends generating a request to a distributed catalog server and processing the response, and S_{QRd} is the mean time the query resolver spends generating a request to a data repository and processing the response.

Each of the queueing service centers also incurs a system overhead in sending and receiving bytes. This overhead includes system call overhead, the costs of transferring each byte of a message from user space to kernel space (or vice versa), and the cost of calculating software checksums in some connection-oriented protocols. S_b is the mean system service time per byte transferred amortized over packets of 1000 data bytes plus lower level protocol overhead. B_{DC} is the total number of bytes in a distributed catalog server request and response (exclusive of the

Quantity	Description
λ	Query arrival rate for all users
S_i	Mean service time of a request at service center $i \in \{DC, DR, NET\}$
S_{QRl}	Mean service time of a meta-data cache lookup at a query resolver
S_{QRc}	Mean service time of sending a catalog request and processing the response at the query resolver
S_{QRd}	Mean service time of sending a data request and processing the response at a query resolver
S_b	Mean system service time of sending or receiving a byte
B_{DC}	Mean number of bytes in a catalog server request and response
B_{DR}	Mean number of bytes in a data request and response
N	Number of data repositories
a	Mean number of attributes in a query
h	Hit rate for the meta-data cache
s	Selection rate for a data repository
I_i	Number of instances of service center $i \in \{DC, DR, QR\}$

Table 7.3. Model inputs

overhead of lower protocols), and B_{DR} is the total number of bytes in a data repository request and response.

N is the number of distinct data repositories (partitions) in the system, and *a* is the average number of attributes in the selection predicate for a query. The hit rate for the meta-data cache, *h*, is the fraction of queries that can be answered without contacting a distributed catalog server for meta-data. The miss rate, $(1 - h)$, is the fraction of queries that miss for some or all of their meta-data cache lookups. Queries on the average contact s percent of the data repositories in the system.

The percentage s is called the *selectivity* of the query. As a simplification, we assume that accesses to data repositories are uniformly distributed. Although an individual user may query some data repositories more often the others, queries from a large number of users are likely to distribute more uniformly. This assumption allows us to conclude that the selectivity of a query is the percentage of all queries that contact any particular data repository. Therefore, the selectivity of a query, s, is also the *selection rate* of a data repository by all queries in the system. If the uniform distribution of queries to data repositories is violated, one or more data repositories become hot spots. A hot spot data repository is accessed by more than λs queries. Data caching reduces the load on hot spot data repositories by caching frequently accessed data locally. Hot spot data repositories can also be replicated to avoid creating a bottleneck in the system.

Our model notation is described in Table 7.4. The service demand, D_i, is the mean time to process a query at service center i. Each query may make multiple requests, or *visits*, to service center i, and each request is processed for service time $S_{i,TOTAL}$. In general, the product of the mean number of visits, V_i, and the mean service time is the mean service demand:

Quantity	Description
D_{DC}	Mean service demand per query at a distributed catalog server
D_{DR}	Mean service demand per query at a data repository
D_{NET}	Mean service demand per query on the network
D_{QR}	Mean service demand per query at a query resolver
R	Mean response time for a query

Table 7.4. Model notation

$$D_i = V_i S_{i,TOTAL}$$

We model the total service time at a distributed catalog server as follows:

$$S_{DC,TOTAL} = S_{DC} + B_{DC} S_b$$

The total service time at the distributed catalog server is the sum of the service time for a request, S_{DC} , and the overhead for transferring the bytes in the request and response, $B_{DC} S_b$. The number of visits to a distributed catalog server follows:

$$V_{DC} = (1 - h) 2a$$

and, therefore, the service demand at a distributed catalog server is:

$$D_{DC} = (1 - h) 2a (S_{DC} + B_{DC} S_b)$$

When the meta-data cache misses for $(1 - h)$ fraction of the queries, the query resolver requests meta-data from a distributed catalog server. A query may miss in the meta-data cache whenever the query resolver follows a path through the global referral graph to constrain an attribute in the query's selection predicate. We use a conservative estimate, $2a$, for the average number of visits to the distributed catalog server.

Our estimate of the number of visits to the distributed catalog server is based on a natural way for structuring the global referral graph. For each path through the global referral graph, the active catalog employs one or more attributes, called *structuring attributes*, to structure the search space by a common classification system for information. Examples of structuring attributes include geographic localities, such as country and city, and organization information, such as organization name and type. The active catalog allows different structuring attributes to be used in different paths through the referral graph. We estimate that there are up to 4 structuring attributes which users supply with queries, because 4 attributes allows considerable flexibility in structuring the search space. For example, having 4 attributes allows us to structure the search space by country, state, city and organization. The referral graphs that we have designed for our prototype use at most 3 structuring attributes (e.g. country, state and city, or country and organization). The rest of the attributes in the query can be used to constrain the search after the referral graph path isolates the query by using the structuring attributes.

If a query has 5 attributes and 3 of them are structuring attributes, then the worst-case traversal of the referral graph will lead to 2 paths of length 4, as shown in Figure 7.6. The length of the path depends on the number of attributes constrained on the path, not on the order of the attributes on the path. In the worst-case, one catalog referral is traversed for each attribute constrained by the path. We can calculate the number of referrals traversed in the referral graph for the number of structuring attributes, a_s, and the number of total attributes, a, in a query. The number of referrals traversed is the product of the path length to constrain each attribute that is not a structuring attribute and the number of attributes that are not structuring attributes: $(a_s + 1)(a - a_s)$. This number is more pessimistic than placing all attributes in the same path, but is more realistic because paths for all possible combinations of attributes are very unlikely. Catalog function maintainers will contribute functions for the smallest set of attributes that are

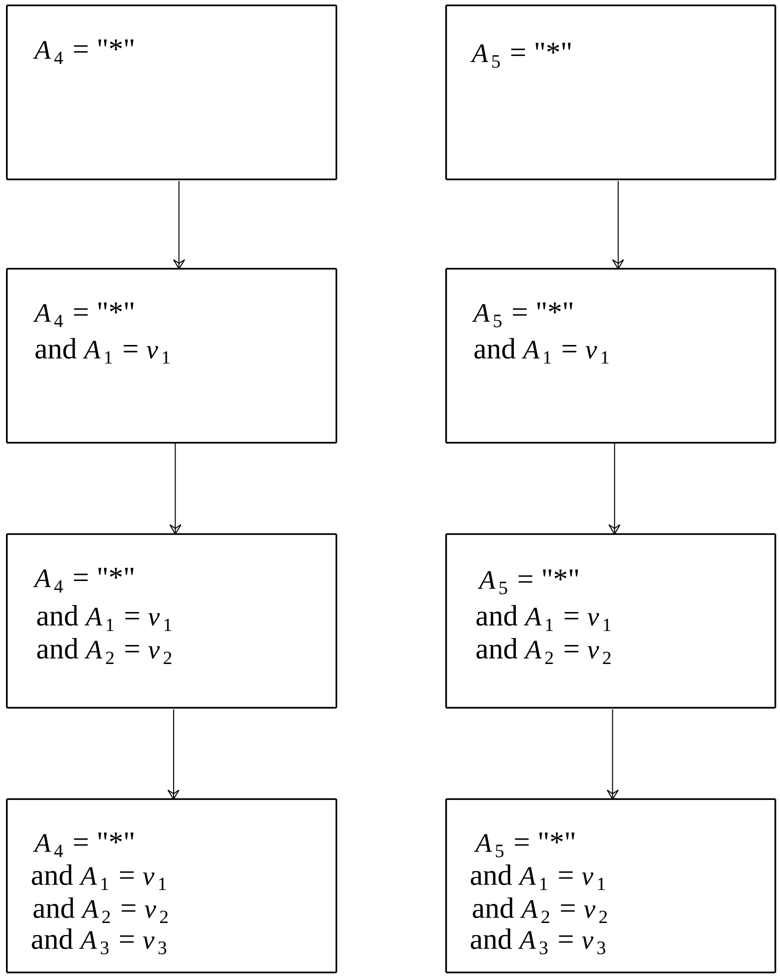


Figure 7.6. Referral graph traversal for a query with 3 structuring attributes and 5 total attributes

(The referrals in the graph are represented by their templates. A_i is the name of an attribute and v_i is the value of the attribute in the query. A_1, A_2, and A_3 are the attributes that structure the search space. The figure depicts the worst-case path length for 3 structuring attributes and 2 additional indexed attributes.)

both known to users and effective in constraining the search, because this technique reduces the work of catalog function creation and maintenance.

For values of a less than 6, and a_s less than 4, the number of referrals traversed is bounded by $2a$. We anticipate that the number of attributes in queries will be a small number less than or equal to 6. If more attributes are supplied, a subset of the attributes can be used to establish the search space. Processing a referral potentially calls a catalog function that must be initialized from the distributed catalog server, so the number of visits to the distributed catalog server is set (pessimistically) to $2a$.

We model the service demand at a data repository as the product of the percentage of queries that visit the data repository and the total service time of the data repository:

$$D_{DR} = s(S_{DR} + B_{DR}S_b)$$

The total service time of one query at a data repository is the sum of the service time for a request, S_{DR}, and the overhead for transferring the bytes in the request and response, $B_{DR}S_b$.

The number of visits made by one query to the network, V_{NET}, is the sum of the number of accesses to a distributed catalog server and data repositories:

$$V_{NET} = (1 - h)2a + Ns$$

The number of data repositories visited by a query is the product of the selectivity and the number of data repositories. The demand on the network is the product of V_{NET} and the service time:

$$D_{NET} = [(1 - h)2a + Ns]S_{NET}$$

Finally, the demand on the query resolver is the sum of the costs of searching the meta-data cache and processing messages to a distributed catalog server and data repositories, as follows:

$$D_{QR} = S_{QRl} + V_{DC}(S_{QRc} + B_{DC}S_b) + Ns(S_{QRd} + B_{DR}S_b)$$

We multiply the number of visits to a distributed catalog server by the total of the service time of the catalog request and data transfer at a query resolver. We multiply the number of data repository visits, Ns, by the total of the service time of the data request and data transfer at a query resolver.

When there is no load on the system, the response time for a query is simply the sum of these demands. We call this response time R_0 and calculate it as follows:

$$R_0 = D_{DC} + ND_{DR} + D_{QR} + D_{NET}$$

We multiply the demand at one data repository by the total number of data repositories in the system to obtain the total demand at data repositories. In the next section, we validate this sum of demands against our previous single user experiments. In subsequent sections, we solve our queueing model and analyze its results.

7.3.2. Measurements

We tested the accuracy of demands in our model by comparing the sum of demands to the response time results of the single user experiments in Section 7.2. In this section, we describe our measurements of the input parameters of the model and the model results for single user response times. We find a high level of agreement between the sum of demands in our model and our experimental measurements of query response time.

Model Input	1 Byte Tuple Measurement	1000 Byte Tuple Measurement
S_{DC}	31	31
S_{DR}	25	26
S_{NET}	2	2
S_{QRl}	7	7
S_{QRc}	26	26
S_{QRd}	24	25
S_b	0.008	0.008
B_{DC}	796	796
B_{DR}	75	1077
N	1000	1000
a	1	1

Table 7.5. Model inputs for the environment of the single user experiments

(Service times in milliseconds. The single user experiments tested the multiple hit and selection rates listed in Table 7.6.)

We measured the input parameters in the environment used for the single user experiments described in Section 7.2. The millisecond service times in Table 7.5 are the averages of thousands of measurements using DECstation 3100s and an Ethernet. The first column in the table describes measurements for tuple sizes of 1 byte, and the second column describes measurements for tuple sizes of 1000 bytes. The multiple hit and selection rates used in the single user experiments are not shown in Table 7.5, but are shown in Table 7.6 which list our comparisons of results.

Tables 7.6a and b report our model and experimental response times in seconds for 1 byte and 1000 byte tuples respectively. The percent difference in the tables is the difference between the response time in the model and the single user experiments as a percentage of the experimental measurement. We compare the model results to the experimental measurements for each combination of selection and hit rates. Figures 7.7 and 7.8 provide graphs of the model and experimental response times for 1 byte and 1000 byte tuples respectively. Since the response times from the model do not differ for cold and warm caches in Nomenclator, we represent the Nomenclator model results with a single line in the graphs.

Selection Rate	Hit Rate	Model Output	Experimental Measurement	Percent Difference
0.00	0	0	1	100
0.25	0	13	15	13
0.50	0	26	30	15
0.75	0	39	38	3
1.00	0	52	50	4
0.00	1	0	0	0
0.25	1	13	14	7
0.50	1	26	28	7
0.75	1	39	38	3
1.00	1	52	50	4

Table 7.6a. Nomenclator model and single user experiment response times for 1 Byte Tuples

(Response times in seconds. Selectivity rate is the percentage of 1000 data repositories contacted by the query. Percent difference is the difference between the response time in the model and the experiments as a percentage of the experimental measurement.)

Tables 7.6a and b show a high level of agreement between our Nomenclator model and experimental measurements. All differences in the tables, except those for the first row of each table, are within 20 percent. The percent difference for the first row is not significant, because the quantities compared are small relative to the other measurements and the rounding of response times to seconds accentuates the difference.

Selection Rate	Hit Rate	Model Output	Experimental Measurement	Percent Difference
0.00	0	0	1	100
0.25	0	18	17	6
0.50	0	35	37	5
0.75	0	53	51	4
1.00	0	70	64	9
0.00	1	0	0	0
0.25	1	18	15	20
0.50	1	35	36	3
0.75	1	53	49	8
1.00	1	70	63	11

Table 7.6b. Nomenclator model and single user experiment response times for 1000 Byte Tuples

(Response times in seconds. Selectivity rate is the percentage of 1000 data repositories contacted by the query. Percent difference is the difference between the response time in the model and the experiments as a percentage of the experimental measurement.)

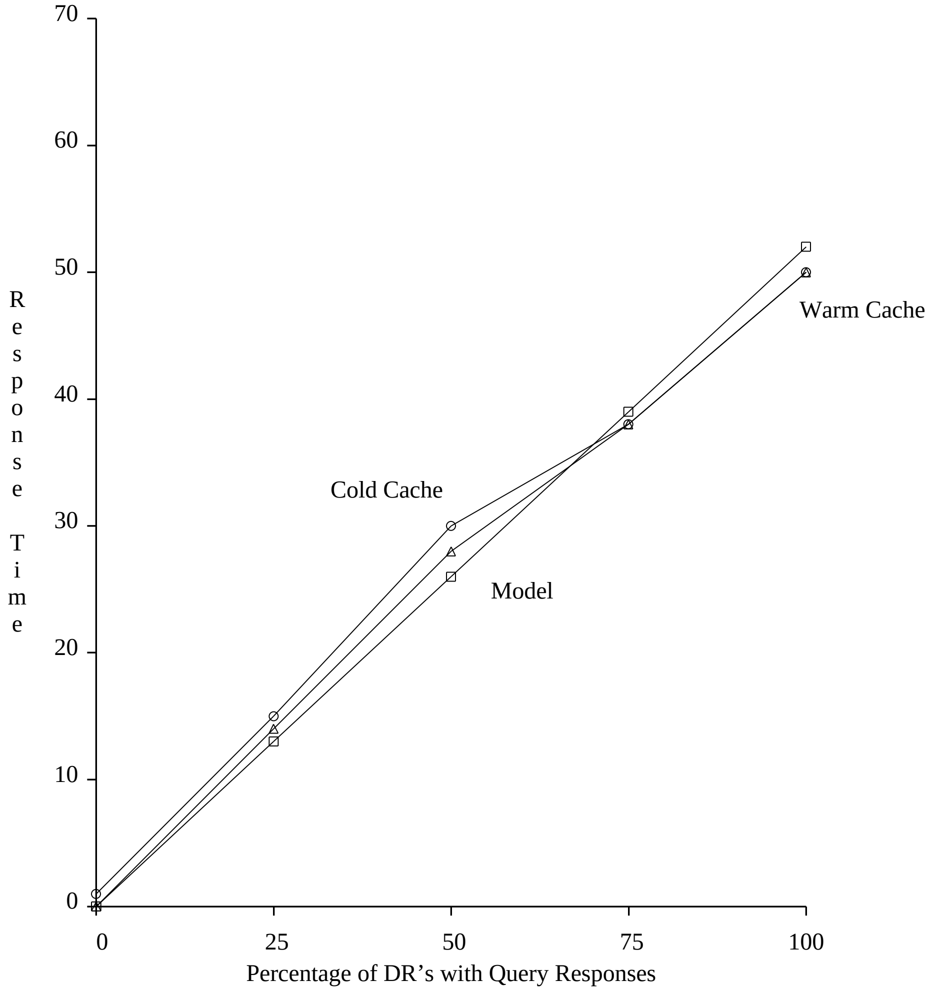


Figure 7.7. Comparison of model and experimental response times for 1 byte response tuples

(Response time in seconds for each data distribution pattern.)

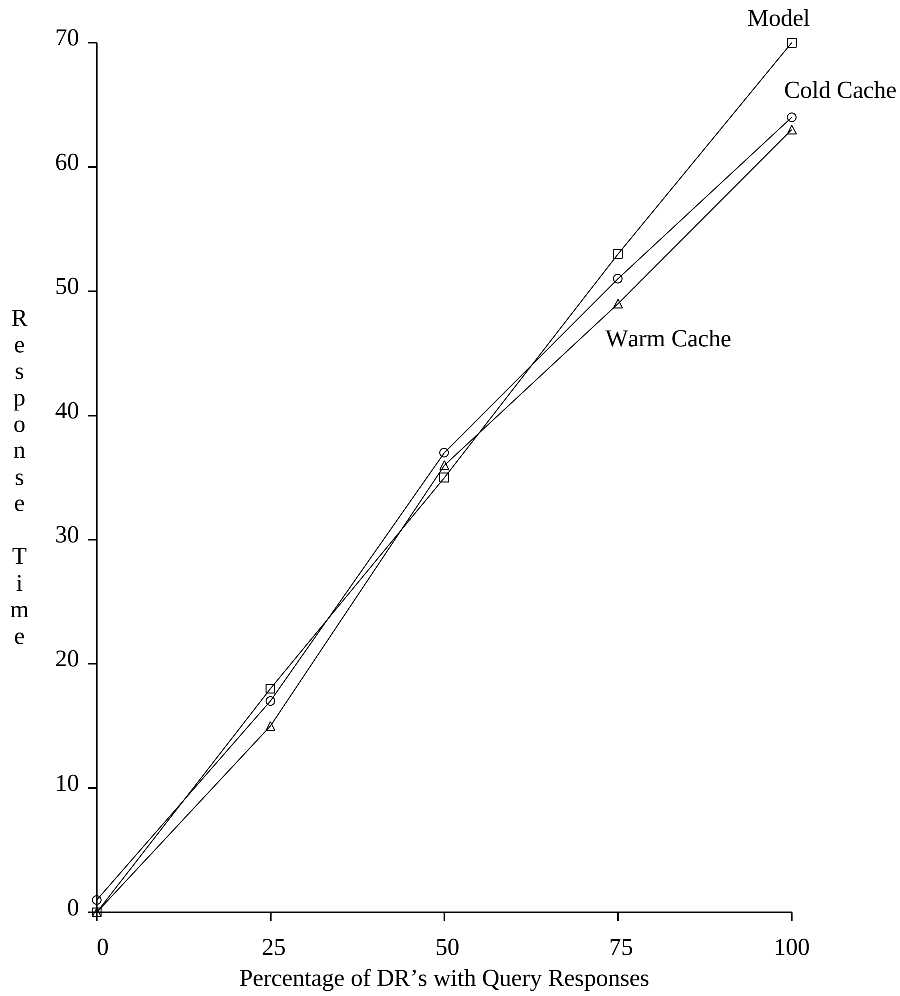


Figure 7.8. Comparison of model and experimental response times for 1000 byte response tuples

(Response time in seconds for each data distribution pattern.)

7.3.3. Model Solution

We obtain the mean response time for queries in the multi-user Nomenclator system by using the solution for open, separable queueing models in [57]. If there is one instance of a query resolver, distributed catalog server, and each data repository in the system, the mean response time for a query is R_1 as follows:

$$R_1 = \frac{D_{DC}}{1 - \lambda D_{DC}} + \frac{ND_{DR}}{1 - \lambda D_{DR}} + \frac{D_{QR}}{1 - \lambda D_{QR}} + D_{NET}$$

λD_i is the utilization at center i, so $(1 - \lambda D_i)$ is the percentage of the service center capacity that is available to process requests. If all the capacity is free, i.e. the denominator is 1 for a particular service center, no queueing occurs and the time spent at the service center is equal to the service demand that the query places on the center. As the percentage of free capacity at the center decreases to 0, the query waits longer for service and time spent at the center grows towards infinity.

Our system is designed so that each organization or department can install and support its own query resolver. We also expect that the distributed catalog server and data repositories will be replicated to support additional query processing load. We add a model input, I_i, to represent the number of instances of service center $i \in \{DC, DR, QR\}$. In general, the mean response time for a query, R, is the following:

$$R = \frac{D_{DC}}{1 - \frac{\lambda}{I_{DC}} D_{DC}} + \frac{ND_{DR}}{1 - \frac{\lambda}{I_{DR}} D_{DR}} + \frac{D_{QR}}{1 - \frac{\lambda}{I_{QR}} D_{QR}} + D_{NET}$$

The fraction $\frac{\lambda}{I_i}$ is the arrival rate of queries at an instance of service center i.

The asymptotic bound on the minimum number of instances needed to service arrival rate λ is:

$$I_{i,\ MIN} = \lambda D_i$$

Table 7.7 summarizes the equations in our queueing model.

7.3.4. Analysis

In this section, we use our model to analyze the number of instances of a system component needed to support descriptive name services and the response times that can be expected in a large internet environment. We characterize a large internet by the input parameters in Table 7.8. S_{DC} is based on our measurements of the prototype implementation of a distributed catalog server, and will probably be closer to S_{DR} in an optimized implementation. S_{DR} is an estimate of the cost of a query in a small memory-resident database. S_{DC} and S_{DR} are the costs of processing the query, without the cost per byte of reading a request and writing a response on the network. S_b, B_{DC}, and B_{DR} quantify these additional costs. S_b is the measured byte service time reported in Section 7.3.2. We estimate the amount of data transferred in a distributed catalog request and response, B_{DC}, to be 8192 bytes. Catalog functions that use more indexing information can be implemented as remote services or retrieve only the portion of the information relevant to the current query. We estimate the amount of data returned by each data repository to be approximately 4 tuples of 500 bytes each plus message overhead. We choose a small average response size from data repositories, because name service users are searching for a few resources or people, not retrieving large volumes of data. B_{DR} is the sum of this response size and the size of a request to a data repository. S_{DC} and S_{DR}, alone, bound

Model Output	Formula
Demand at the distributed catalog	$D_{DC} = (1 - h)\,2a\,(S_{DC} + B_{DC}S_b)$
Demand at a data repository	$D_{DR} = s\,(S_{DR} + B_{DR}S_b)$
Demand on the network	$D_{NET} = [\,(1 - h)\,2a + Ns\,]\,S_{NET}$
Demand at the query resolver	$D_{QR} = S_{QRl} + (1 - h)\,2a\,(S_{QRc} + B_{DC}S_b) + Ns\,(S_{QRd} + B_{DR}S_b)$
Query response time	$R = \dfrac{D_{DC}}{1 - \dfrac{\lambda}{I_{DC}} D_{DC}} + \dfrac{ND_{DR}}{1 - \dfrac{\lambda}{I_{DR}} D_{DR}} + \dfrac{D_{QR}}{1 - \dfrac{\lambda}{I_{QR}} D_{QR}} + D_{NET}$
Instances of queueing center i required to service λ	$I_{i,\,MIN} = \lambda D_i$

Table 7.7. Nomenclator model summary

query throughput at 32 and 38 queries per second. Adding data transfer costs reduces the throughput of a distributed catalog server to 10 queries per second, and of a data repository to 24 queries per second.

We increase the network service time measured on our local area network to 200 milliseconds to represent the additional delay of wide-area communication. This network service time is 4 times the measured round-trip delay for East to West Coast access on an idle network in the United States. Experiments with `ping` indicate that

Model Input	Input Value
S_{DC}	31
S_{DR}	26
S_{NET}	200
S_{QRl}	200
S_{QRc}	26
S_{QRd}	25
S_b	0.008
B_{DC}	8,192
B_{DR}	2,048
N	10,000

Table 7.8. Model inputs for a large internet

(Service times in milliseconds. We analyze the effects of multiple values of the remaining model inputs in subsequent figures.)

round-trip times from Wisconsin to Japan on busy networks are within 200 milliseconds, and closer systems have significantly lower round-trip times. We set S_{QRl} to the time we measured in our prototype for searching a meta-data cache of 1000 entries. S_{QRc} and S_{QRd} are estimates of the time for processing a distributed catalog request and a data repository request respectively, based on measurements from our prototype query resolver implementation. The Internet currently has approximately 10,000 organizations, so we set the number of data repositories to 10,000[1]. We vary *s, h, a* and λ during the following model analysis.

Our target query workload contains queries that can be constrained to a few hundred data repositories. Queries that search more data repositories are possible, but they will be used by a small (desperate) percentage of the user population. In this analysis, we investigate selectivities that search 100 or 300 of the data repositories. We consider system-wide query arrival rates of 20, 200, 1000, and 2000 queries per second. One million users making 12 queries per week produces a arrival rate of 20 queries per second, and one million users making 120 queries per week produces an arrival rate of 200 queries per second. Six million users asking 100 queries per week produce

[1] The number of organizations as indicated by the number of domains available for zone transfer in the Domain Name System reached 10,000 in 1991 [63] and 20,000 in 1993 [64].

1000 queries per second, and 12 million users asking 100 queries per week produce 2000 queries per second. The White Pages Workshop challenged future name services to support arrival rates at the lower end of this range [104].

We report two types of results from our model. First, we use $I_{i,\ MIN}$ to calculate the number of instances of system components needed to support the arrival rates in our target environment. Second, we configure a system with the recommended number of instances for an arrival rate of 200 queries per second, and analyze the response time results. We choose 200 queries per second, because it exceeds the requirements of the White Pages Workshop and provides a generous configuration for 1 million users. One million users can make more than 100 queries per week with this configuration.

The number of replicas of the distributed catalog server, $I_{DC,\ MIN}$, depends on the values of a, h and λ. Figure 7.9 graphs the number of replicas of the distributed catalog server as a function of the meta-data cache miss rate $(1 - h)$ and λ for an average of 4 attributes per query ($a = 4$). $I_{DR,\ MIN}$ depends only on the selection rate s and λ. Figure 7.10 graphs the number of replicas of a data repository as a function of these inputs. $I_{QR,\ MIN}$ depends on the values of s, λ, h and a. Figure 7.11 graphs the number of instances of the query resolver as a function of the selection rate and λ for a conservative miss rate of 100% and an average of 4 attributes per query.

Figure 7.9 shows the benefit of meta-data caching in scaling the distributed catalog service to many users. As meta-data cache misses decrease, the number of replicas of the distributed catalog server is reduced. Given the previous success of caching in name services, we expect a miss rate of 30% for meta-data caching. As we will show during our discussion of Figure 7.11, it is advantageous for each department or group in an organization to have its own instance of a query resolver. Overlapping interests and frequent collaborations between members of the department with a small number of other organizations will bring meta-data into the cache, and the meta-data will be reused. At a 30% miss rate, approximately 250 distributed catalog server replicas are needed to support 1000 queries per second, or 1 replica for every 24,000 users. The replicas required represent only 0.004% of the size of the user population.

Figure 7.10 shows the benefit of the active catalog in scaling data repositories to many users. As queries become more selective, the data repositories can support more users without replication. For queries constrained to search a few hundred data repositories, 3 or 4 instances of each data repository are required for all the arrival rates in the graph. Data owners can make their information accessible and protect their data repositories from the excessive load of brute force searches by supplying meta-data to the active catalog.

Figure 7.11 shows that approximately 20,000 query resolver instances are required to support 6 million users asking 100 queries per week ($\lambda = 1000$) when the queries are constrained to 500 or fewer data repositories ($s = 0.05$). One query resolver is required for every 300 users, i.e. one query resolver is required to service the requests of a facility the size of the University of Wisconsin Computer Sciences Department. Users can support their own query resolver, and add query resolvers as their use of the system increases. The ability to support 300 users allows larger, shared meta-data caches that can improve the miss rate. Requiring one query resolver per department is similar to the current requirements of the Domain Name Service.

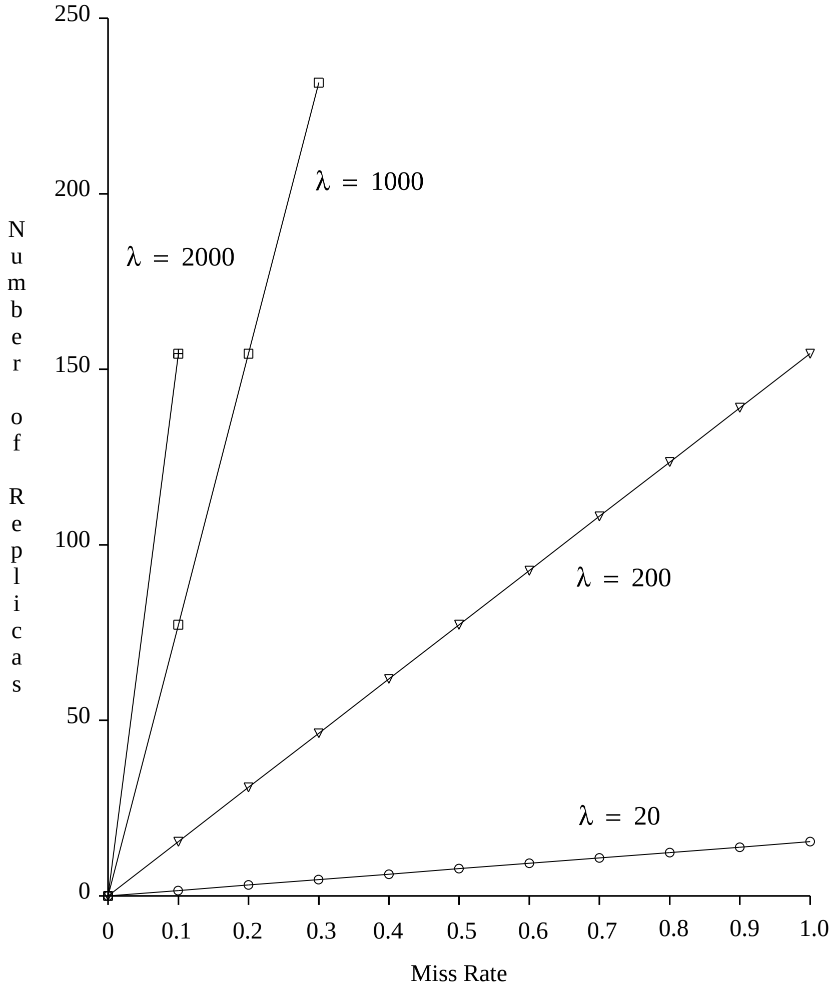


Figure 7.9. Distributed Catalog Server Replicas

(The minimum number of replicas of a distributed catalog server needed to service arrival rate λ for different miss rates and an average of 4 attributes per query. λ is in queries per second.)

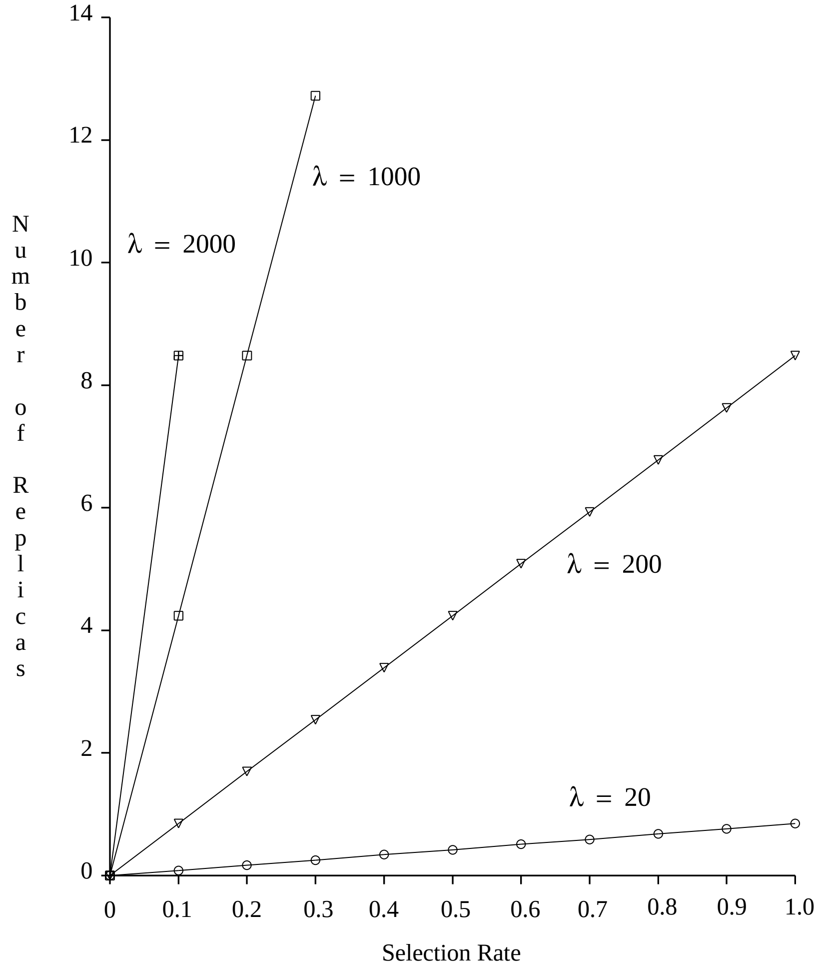


Figure 7.10. Data Repository Replicas

(The minimum number of replicas of each data repository needed to service arrival rate λ for different selectivity rates. λ is in queries per second.)

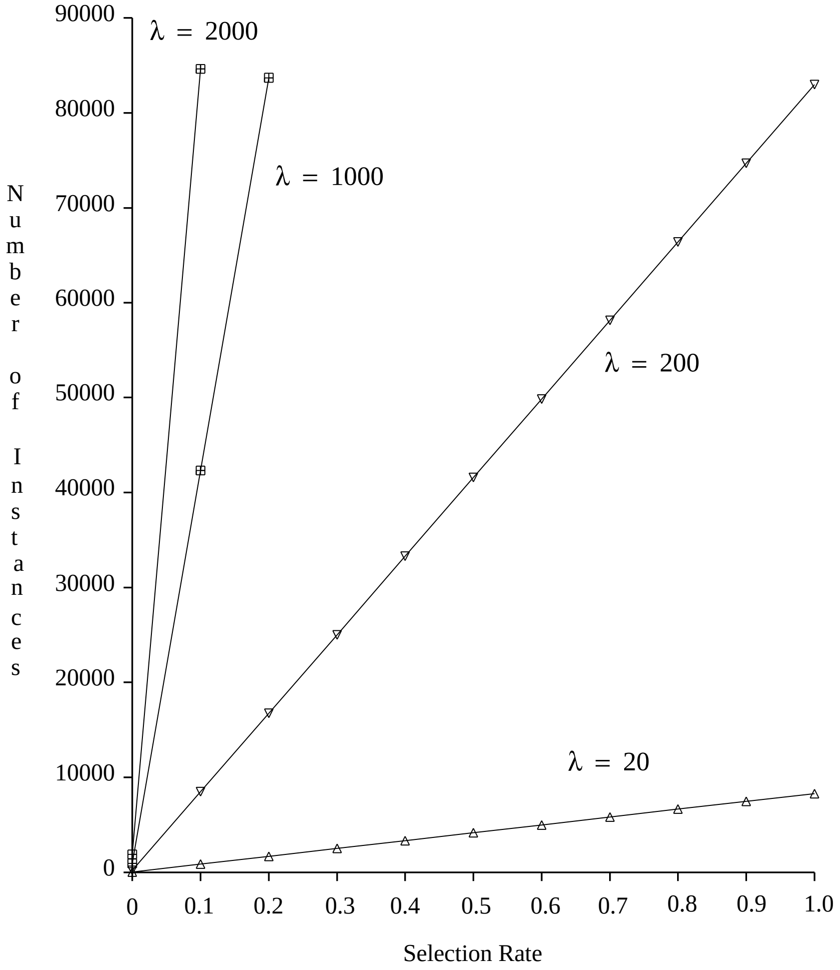


Figure 7.11. Query Resolver Instances

(The minimum number of instances of a query resolver needed to service arrival rate λ for different selectivity rates, a conservative miss rate of 100%, and an average of 4 attributes per query. λ is in queries per second.)

We calculated average response times, *R*, for the Internet parameters in Table 7.8. We used 10,000 instances of query resolvers, 1 instance of each data repository, and 155 replicas of the distributed catalog server to calculate the response times. We chose the number of replicas for the distributed catalog server from the results displayed graphically in Figure 7.9 to support 200 queries for any meta-data cache miss rate. The replicas also support 1000 queries per second at a 20% miss rate, and 2000 queries per second at a 10% miss rate. The number of query resolvers is set to reflect the number of organizations currently participating in the Internet. The owners of data are not required to provide replicas of data repositories by this configuration, so the system could be phased into the existing Internet.

Figure 7.12 reports the response time results for queries that search 100 or 300 data repositories at miss rates of 10% and 30%. Figure 7.13 reports the demand per instance of the distributed catalog server, the query resolver,

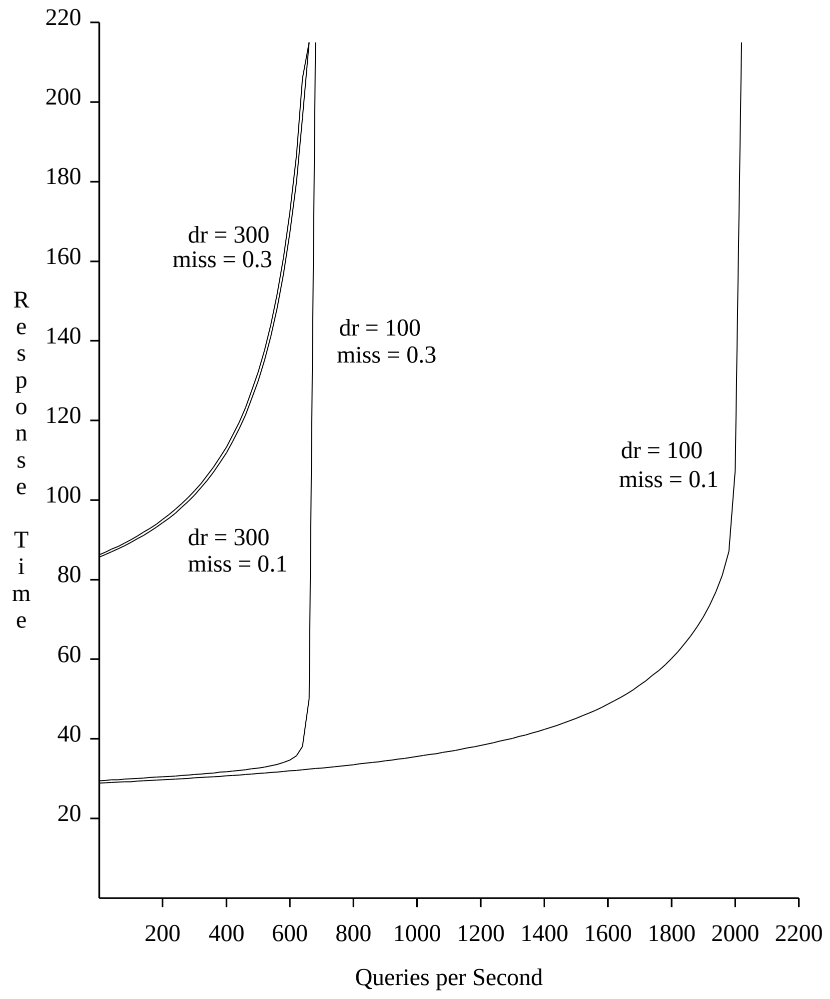


Figure 7.12. Model response time results

(Response times in seconds and λ in queries per second. Response times are reported for searches constrained to 100 or 300 data repositories ($s = 0.01$ or $s = 0.03$), and for miss rates of 10% or 30%. The system includes 155 replicas of the distributed catalog server, one instance of each data repository, and 10,000 instances of query resolvers. Queries contain an average of 4 attributes.)

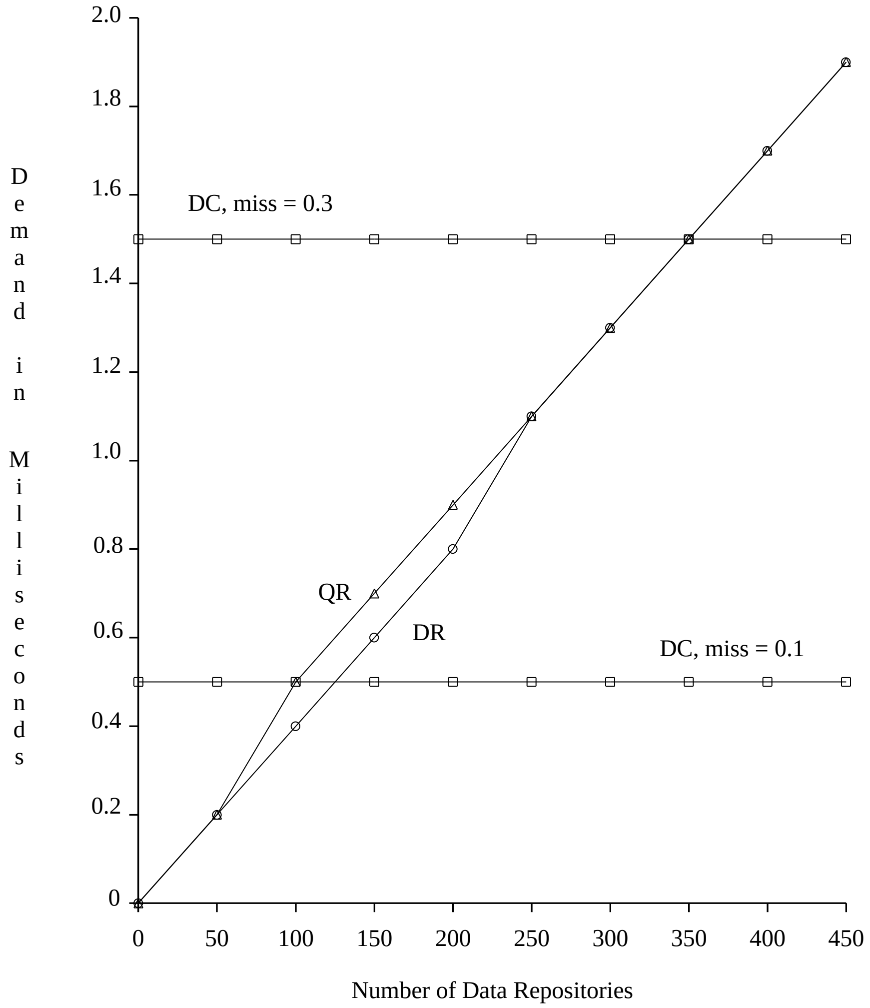


Figure 7.13. Demand per instance of a service center

(Demand per query in milliseconds for each instance of a distributed catalog server (DC), data repository (DR), and query resolver (QR) for miss rates of 10% and 30%. The system includes 155 replicas of the distributed catalog server, one instance of each data repository, and 10,000 instances of query resolvers. Queries contain an average of 4 attributes.)

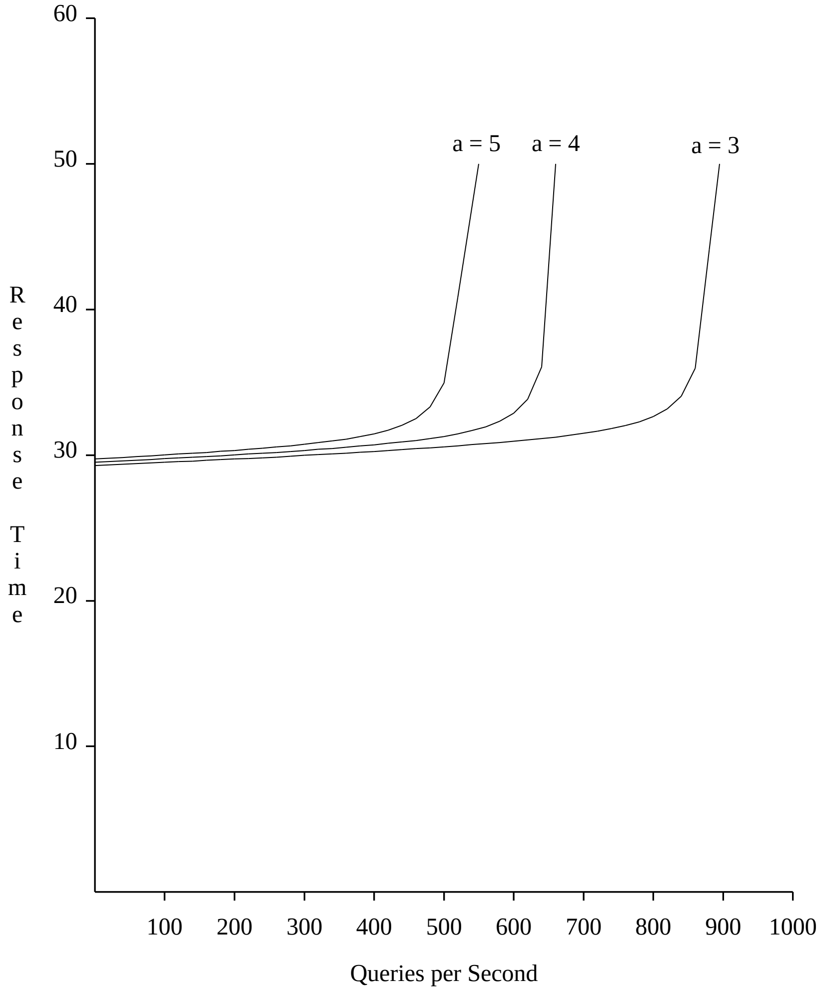


Figure 7.14. Response time results for different values of *a*

(Response times in seconds and λ in queries per second. Response times are reported for different numbers of attributes, a, in a query. Each query is constrained to 100 data repositories ($s = 0.01$) and has a miss rate of 30%. The system includes 155 replicas of the distributed catalog server, one instance of each data repository, and 10,000 instances of query resolvers.)

and a data repository for miss rates of 10% and 30%. The demand per instance is the demand (i.e. D_{DC}, D_{DR}, or D_{QR}) of one query uniformly distributed over the instances of the each resource. For example, the total demand of a query on the distributed catalog servers is the number of replicas (155) times the demand displayed on the graph. The demand at each data repository instance is represented by a single line on the graph, because the demand is independent of the miss rate. The demand at each query resolver instance is insensitive to the miss rates, varying only by 0.02 ms, so it is also represented by a single line on the graph. We set *a* to 4 for each of these figures.

Figure 7.14 compares response time behavior for three different values for *a*.

Figure 7.12 shows a response time of 30 seconds for searches of 100 data repositories, and 90 seconds for searches of 300 data repositories. These numbers represent the order of magnitude of response times in a large Internet environment, so it is reasonable to expect response times of a few minutes for our query workload. Query performance for searches of 100 data repositories reaches capacity at 600 queries per second when the miss rate is 30%, because the distributed catalog servers saturate. In the other cases, performance degrades more gradually as multiple resources saturate. Figure 7.13 confirms this analysis. In Figure 7.13, the demand at the distributed catalog server far exceeds the demand at the query resolvers and data repositories for searches of 100 data repositories and a miss rate of 30%. The three demands are comparable when the miss rate is 10%. The demand at the query resolvers and data repositories exceeds the distributed catalog demand for searches of 300 data repositories and a miss rate of 10%, and the demands are comparable when the miss rate is 30%. The sharp increase in response times when the distributed catalog service saturates alone, in comparison to the more gradual increase when data repositories or query resolvers also saturate, indicates that the utilization of the distributed catalog service should be monitored so servers can be replicated before performance abruptly decays.

Figure 7.14 shows how the number of attributes in a query affects the capacity of our system. When the distributed catalog servers are the bottleneck to system performance, more attributes in the query reduce system capacity. Queries with more attributes may visit the distributed catalog server more times on a meta-data cache miss. Since the percentage of a distributed catalog server needed to support each user is small, the disadvantages of more attributes in a query are offset by their advantages in constraining the search.

7.4. Summary

In this chapter, we described the results of three performance studies of Nomenclator. Our first two studies showed how our techniques reduce the impact of the individual user on the query processing environment. Our final study showed how our techniques support scaling in multi-user environments. We showed that the performance of descriptive queries can be improved by constraining the search with an active catalog and by caching meta-data in an X.500 environment. We showed that active cataloging is the most significant contributor to better query response time in a system with low load, while meta-data caching functions to reduce the load on the system. We showed that meta-data caching plays a significant role in scaling the system to many users by reducing the number of distributed catalog servers needed to support large workloads. By our calculations, six million users asking 100 queries per week require a number of replicas that is just 0.004% of the size of the user population. Query resolvers are designed to be run at the level of a department or division in an organization. When query resolvers are created in this way, and the distributed catalog servers and data repositories are adequately replicated, the system scales successfully to millions of users.

Chapter 8

CONCLUSIONS AND FUTURE WORK

8.1. Conclusions

This thesis shows that the key to fast descriptive name services is constraining the search space to a small number of data repositories. Previous descriptive name services have not scaled to thousands of data repositories and millions of users, because they rely on exhaustive search techniques. Constraining the search space improves response time by eliminating the work of contacting data repositories that do not contribute to the query answer. It also improves scaling by reducing the load on data repositories, query resolvers and the network.

We provide two new techniques for constraining the search space: an active catalog and meta-data caching. The active catalog constrains the search space for a query by returning a list of data repositories where the answer to the query is likely to be found. Components of the catalog, called catalog functions, are distributed indices that isolate queries to parts of the network, and smart algorithms for limiting the search space by using semantic, syntactic, or structural constraints. Data owners are motivated to contribute catalog functions, because the catalog functions make the information in their data repositories available to users while reducing the load on the data repositories from user queries.

Information in the active catalog is intelligently replicated in meta-data caches to tailor query sites to the types of queries they see most frequently. Intelligent replication is a partial replication; no one site contains the entire contents of the active catalog but rather those parts that are currently most useful to it. Information on which catalog functions to use, catalog functions and the constrained search spaces that result from using catalog functions are cached for subsequent use. Meta-data caching improves performance by keeping frequently used characterizations of the search space close to the user, and eliminating active catalog communication and processing costs. Meta-data caching provides a scalable alternative to the centralized index servers currently used in Archie and other systems.

Catalog functions and their results can be re-used, because referrals describe the conditions for re-using them. Referrals form a DAG, called a referral graph, that integrates access functions into a uniform query processing framework. The partial order that defines the graph directs query processing by identifying which referrals will constrain the search space further. The referral graph also identifies search spaces that can be combined through intersection and union. The referral graph and four simple meta-data rules for its construction (summarized in Table 4.1) allow us to unify a wide variety of indexing techniques. Catalog functions contributed by different organizations can be integrated into one structure to speed query processing for everyone. Referrals and the meta-data rules also unite our meta-data caching techniques with popular data caching techniques.

We present three performance studies of our system. In the first study, we showed that we can use an active catalog and meta-data caching to constrain the search effectively in a real environment, the X.500 name space. Our experiments consistently showed that meta-data caching and an active catalog provide an effective tool to improve X.500 performance. In the second study, we examined the performance of an active catalog and meta-data caching for single users on a local area network. Our experiments showed that our techniques to eliminate data repositories from the search space can dramatically improve response time. The improvement exists, because latency reduction is critical to large-scale name service query optimization. We showed that active cataloging is the most significant contributor to better response time in a system with low load, and that meta-data caching functions to reduce the load on the system. In the third study, we used an analytical model to evaluate the performance and scaling of our techniques for a large Internet environment. We showed that our system scales well to millions of users for a small investment in distributed catalog servers, one query resolver per department or small organization, and a few data repository replicas. For an environment with a meta-data cache hit rate of 70%, queries that search a few hundred data repositories, and 6,000,000 users asking 100 queries per week, we need, by our calculations, a distributed catalog server for every 24,000 users (just 0.004% of the number of users), a query resolver for every 300 users, and a few replicas of each data repository. This investment is small in comparison to the benefits of fast descriptive name services.

8.2. Future Work

The need for an investment in distributed catalog servers to support descriptive naming highlights an important area for future research. Much research focuses on developing the hardware and software resources to support high-speed networks and distributed multi-media applications. Governments and organizations invest in substantial equipment and software for these networks. The results of this thesis indicate that an additional investment must be made in deploying resources to support an information infrastructure to locate, process and manage the information resources on these networks. We recommend the allocation of servers to support wide-area descriptive name services. This thesis suggests that other information-based network services, like network management, collaborative work, information retrieval, and information filtering, will require similar investments in resources. Research must identify the shared requirements of these systems and develop an information infrastructure that will be intrinsically tied to the computer networks of the future.

This thesis addresses issues of scale and performance. In the short term, the work in this thesis can be extended by workload studies, automating active catalog maintenance, automating access function construction, and supporting updates. First, since we have made large-scale descriptive name services possible, an important next step is to use the service to support a large user community on the Internet. Studies of the user workload will supply important information for studying and improving the performance characteristics of descriptive name services and other large-scale information services. Second, Chapter 6 suggests an approach to maintaining the active catalog. This work can be extended by the formal definition of maintenance algorithms for the active catalog, evaluations of the performance of the algorithms, and formal proof that the algorithms preserve referral graph invariants. Third, we wrote all our access functions directly in C. Additional work is needed to automate access function construction, perhaps by creating a toolkit that allows users to define functions by specifying the characteristics of indices and data. Fourth, future work must identify the best algorithms for maintaining the active catalog for various update workloads. Although name services rarely require large-scale updates, one relational interface for updating collections of component name services would also be useful. Administrators could maintain replicas across several name services, making their naming data available to different user communities. Control over updates would also help Nomenclator to provide more accurate catalog functions and caches.

In the long term, the work in this thesis can be extended by parallel processing and mobile computing support, and by addressing scaling issues for other applications. First, in the wide area environment, programs experience bottlenecks in sending information requests to thousands of sites. Even if we had multicast services, programs would experience bottlenecks in processing responses from a multicast message. We need mechanisms that will distribute queries to a dynamically chosen group of servers, coalesce responses and return results with high levels of parallelism.

Second, issues of scale and performance in name services must be revisited for the new mobile computing environment. There are two basic solutions to large scale mobile computing. One uses constant host addresses, and the other uses variable host addresses and constant names. Cache performance issues arise in the routing layer (with constant host addresses) or in the name service (with constant names), because the frequency and volume of updates increase over previous caching schemes. In both cases, queries about resources in a geographic locality require new cooperation between the routing layer and the name service.

Third, other distributed information services, such as information retrieval, information filtering, and distributed databases, must be moved into large Internet environments. Information retrieval and filtering differ from name services, because they index full documents that may have less selective keywords than those associated with naming data. Distributed databases differ, because they have stricter consistency requirements and a wider range of queries that includes join queries. Active cataloging, meta-data caching, and perhaps other scaling techniques must be extended to these applications to make global information services a reality.

REFERENCES

[1] Accetta, M., ‘‘Resource Location Protocol,’’ Request for Comments 887 (December 1983).

[2] Alonso, R., D. Barbara, and H. Garcia-Molina, ‘‘Data Caching Issues in an Information Retrieval System,’’ *ACM Transactions on Database Systems* **15**(3), pp. 359-384 (September 1990).

[3] Anklesaria, F., M. McCahill, P. Lindner, D. Johnson, D. Torrey, and B. Alberti, ‘‘The Internet Gopher Protocol,’’ Request for Comments 1436 (March 1993).

[4] Babb, E., ‘‘Implementing a Relational Database by Means of Specialized Hardware,’’ *ACM Transactions on Database Systems* **4**(1), pp. 1-29 (March 1979).

[5] Barbara, D., ‘‘Extending the Scope of Database Services,’’ *ACM SIGMOD Record* **22**(1), pp. 68-73 (March 1993).

[6] Batini, C., M. Lenzerini, and S. B. Navathe, ‘‘A Comparative Analysis of Methodologies for Database Schema Integration,’’ *ACM Computing Surveys* **18**(4), pp. 323-364 (December 1986).

[7] Berners-Lee, T., R. Caillau, J.-F. Groff, and B. Pollermann, ‘‘World-Wide Web: The Information Universe,’’ *Electronic Networking: Research, Applications and Policy* **1**(2), Meckler, Westport, CT(Spring 1992).

[8] Birrell, A. D., 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).

[9] Bolot, J.-C. and H. Afifi, ‘‘Evaluating Caching Schemes for the X.500 Directory System,’’ *Thirteenth International IEEE Conference on Distributed Computing Systems*, Pittsburgh, PA, pp. 112-119 (May 1993).

[10] Bowman, C. M., P. B. Danzig, and M. F. Schwartz, ‘‘Research Problems for Scalable Resource Discovery,’’ *Internet Society INET ’93 Conference*, San Francisco, pp. DFB1-DFB10 (August 1993).

[11] Bowman, C. M. and C. Dharap, ‘‘The Enterprise Distributed White-Pages Service,’’ *Usenix Conference*, San Diego, pp. 349-360 (Winter 1993).

[12] Bowman, M., L. Peterson, and A. Yeatts, ‘‘Univers: An Attribute-Based Name Server,’’ *Software – Practice and Experience* **20**(4), pp. 403-424 (April 1990).

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

[14] Breitbart, Y., A. Silberschatz, and G. R. Thompson, ‘‘Reliable Transaction Management in a Multidatabase System,’’ *ACM SIGMOD International Conference on Management of Data*, Atlantic City, pp. 215-224 (May 1990).

[15] Cate, V., ‘‘Alex – a Global Filesystem,’’ *Usenix Workshop on File Systems*, Ann Arbor, MI, pp. 1-11 (May 1992).

[16] Cheriton, D. and T. P. Mann, ‘‘Decentralizing a Global Naming Service for Improved Performance,’’ *ACM Transactions on Computer Systems* **7**(2), pp. 147-183 (May 1989).

[17] Cheriton, D. R. and T. P. Mann, ‘‘Uniform Access to Distributed Name Interpretation in the V-System,’’ *Fourth International IEEE Conference on Distributed Computing Systems*, San Francisco, CA, pp. 290-297 (May 1984).

[18] Clifton, C. and H. Garcia-Molina, ‘‘Distributed Processing of Filtering Queries in HyperFile,’’ *Eleventh International IEEE Conference on Distributed Computing Systems*, Arlington, Texas, pp. 54-64 (May 1991).

[19] Cohrs, D. L., B. P. Miller, and L. A. Call, ‘‘Distributed Upcalls: A Mechanism for Layering Asynchronous Abstractions,’’ *Eighth International IEEE Conference on Distributed Computing Systems*, San Jose, CA, pp. 55-62 (June 1988).

[20] Comer, D. E. and T. P. Murtagh, ‘‘The Tilde File Naming Scheme,’’ *Sixth International IEEE Conference on Distributed Computing Systems*, Cambridge, MA, pp. 509-514 (May 1986).

[21] Comer, D. E., R. E. Droms, and T. P. Murtagh, ‘‘An Experimental Implementation of the Tilde Naming System,’’ *Computing Systems* **4**(3), pp. 487-515 (Fall 1990).

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

[23] Cooper, E. C., ‘‘Replicated Distributed Programs,’’ *Tenth ACM Symposium on Operating Systems Principles*, Orcas Island, WA, pp. 63-78 (December 1985).

[24] (DEC), Digital Equipment Corporation, ‘‘DNA Naming Service Functional Specification Version 1.0.1,’’ Order no. EK-DNANS-FS-001 (November 1988).

[25] Craft, D. H., ‘‘Resource Management In a Decentralized System,’’ *Ninth ACM Symposium on Operating Systems Principles*, Bretton Woods, NH, pp. 11-19 (October 1983).

[26] Danzig, P. B., K. Obraczka, and A. Kumar, ‘‘An Analysis of Wide Area Name Server Traffic,’’ *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, Baltimore, pp. 281-292 (August 1992).

[27] Danzig, P. B., S.-H. Li, and K. Obraczka, ‘‘Distributed Indexing of Autonomous Internet Services,’’ *Computing Systems* **5**(4), pp. 433-460 (1992).

[28] Danzig, P. B., K. Obraczka, and S.-H. Li, ‘‘Internet Resource Discovery Services,’’ *Computer* **26**(9), pp. 8-22 (September 1993).

[29] Davidson, S. B., H. Garcia-Molina, and D. Skeen, ‘‘Consistency in Partitioned Networks,’’ *Computing Surveys* **17**(3), pp. 341-370 (September 1985).

[30] Demers, A., D. Greene, A. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, ‘‘Epidemic Algorithms for Replicated Database Maintenance,’’ *Sixth ACM Symposium on the Principles of Distributed Computing*, Vancouver, pp. 1-12 (August 1987).

[31] Deutsch, P. and et. al., ‘‘Architecture of the WHOIS++ Service,’’ Internet Draft, Merit Network, Inc. (September 1993).

[32] DeWitt, D., 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).

[33] Droms, R. E., ‘‘The Knowbot Information Service,’’ Internet Draft, Bucknell University (December 1989).

[34] Droms, R. E., ‘‘Access to Heterogeneous Directory Services,’’ *IEEE INFOCOM ’90*, San Francisco, pp. 1054-1061 (June 1990).

[35] Emtage, A. and P. Deutsch, ‘‘archie – An Electronic Directory Service for the Internet,’’ *Usenix Conference*, San Francisco, pp. 93-110 (Winter 1992).

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

[37] Fischer, M. J. and A. Michael, ‘‘Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network,’’ *ACM Symposium on Principles of Database Systems*, Los Angeles, pp. 215-224 (March 1982).

[38] Floyd, R. A. and C. S. Ellis, ‘‘Directory Reference Patterns in Hierarchical File Systems,’’ *IEEE Transactions on Knowledge and Data Engineering* **1**(2), pp. 238-247 (June 1989).

[39] Garcia-Molina, H. and G. Wiederhold, ‘‘Read-Only Transactions in a Distributed Database,’’ *ACM Transactions on Database Systems* **7**(2), pp. 209-234 (June 1982).

[40] Gifford, D. K., R. W. Baldwin, S. T. Berlin, and J. M. Lucassen, ‘‘An Architecture for Large Scale Information Systems,’’ *Tenth ACM Symposium on Operating Systems Principles*, Orcas Island, WA, pp. 161-169 (December 1985).

[41] Gifford, D. K., P. Jouvelot, M. A. Sheldon, and J. W. O’Toole, Jr., ‘‘Semantic File Systems,’’ *Thirteenth ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, pp. 16-25 (October 1991).

[42] Hardcastle−Kille, S. E., ‘‘Using the OSI Directory to Achieve User Friendly Naming,’’ Request for Comments 1484 (July 1993).

[43] Hardy, D. R. and M. F. Schwartz, ‘‘Essence: A Resource Discovery System Based on Semantic File Indexing,’’ *Usenix Conference*, San Diego, pp. 361-373 (Winter 1993).

[44] Harrenstien, K., M. Stahl, and E. Feinler, ‘‘NICNAME/WHOIS,’’ Request for Comments 954 (October 1985).

[45] Hauzeur, B. M., ‘‘A Model for Naming, Addressing, and Routing,’’ *ACM Transactions on Office Information Systems* **4**(4), pp. 293-311 (October 1986).

[46] Huitema, C., ‘‘The X.500 Directory Services,’’ *Fourth European Networkshop*, Les Diablerets, Switzerland, pp. 161-166 (May 1988). Published as *Computer Networks and ISDN Systems* **16** (1988).

[47] Jakobs, K., ‘‘The Directory - Evolution of a Standard,’’ *IFIP TC6/TC8 Open Symposium on Network Information Processing Systems*, Sofia, Bulgaria, pp. 281-289 (May 1988).

[48] Kahle, B., ‘‘Wide Area Information Server Concepts,’’ Technical Report TMC-202, Thinking Machines Corporation, Cambridge, MA (August, 1990).

[49] Kahn, R. E. and V. G. Cerf, ‘‘The Digital Library Project,’’ Technical Report, Corporation for National Research Initiatives, Reston, VA (March 1988).

[50] Kernighan, B. W. and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ (1988).

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

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

[53] Kim, W. and J. Seo, ‘‘Classifying Schematic and Data Heterogeneity in Multidatabase Systems,’’ *Computer* **24**(12), pp. 12-18 (December 1991).

[54] Knuth, D. E., *The Art of Computer Programming*, Addison-Wesley, Reading, MA (1973).

[55] Lampson, B. W., ‘‘Designing a Global Name Service,’’ *Fifth ACM Symposium on Principles of Distributed Computing*, Calgary, Alberta, Canada, pp. 1-10 (August 1986).

[56] Landweber, L., M. Litzkow, D. Neuhengen, and M. Solomon, ‘‘Architecture of the CSNET Name Server,’’ *SIGCOMM Symposium on Communications Architectures and Protocols*, Austin, pp. 146-149 (March 1983).

[57] Lazowska, E. D., J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance*, Prentice−Hall, Englewood Cliffs, NJ (1984).

[58] Leffler, S., M. Karels, and M. McKusick, ‘‘Measuring and Improving Performance of 4.2BSD,’’ *Usenix Conference*, Pittsburgh, pp. 213-225 (Summer 1984).

[59] Levy, E. and A. Silberschatz, ‘‘Distributed File Systems: Concepts and Examples,’’ *ACM Computing Surveys* **22**(4), pp. 321-374 (December 1990).

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

[61] Lindsay, B. G., ‘‘Object Naming and Catalog Management for a Distributed Database Manager,’’ *Second International IEEE Conference on Distributed Computing Systems*, Paris, pp. 31-40 (April 1981).

[62] Litwin, W., L. Mark, and N. Roussopoulos, ‘‘Interoperability of Multiple Autonomous Databases,’’ *ACM Computing Surveys* **22**(3), pp. 267-293 (September 1990).

[63] Lottor, M., ‘‘Internet Growth (1981-1991),’’ Request for Comments 1296 (January 1992).

[64] Lottor, M., ‘‘Internet Domain Survey,’’ Network Information Systems Center Report, SRI International (January 1993).

[65] Miller, R. J., Y. E. Ioannidis, and R. Ramakrishnan, ‘‘The Use of Information Capacity in Schema Integration and Translation,’’ *Nineteenth International Conference on Very Large Data Bases*, Dublin, pp. 120-133 (August 1993).

[66] Mockapetris, P. V., ‘‘The Domain Name System,’’ *IFIP WG 6.5 Working Conference on Computer-Based Message Services*, Nottingham, England, pp. 61-72 (May 1984).

[67] Mockapetris, P. V., ‘‘Domain Names - Concepts and Facilities,’’ Request for Comments 1034 (November 1987).

[68] Mockapetris, P. V., ‘‘Domain Names - Implementation and Specification,’’ Request for Comments 1035 (November 1987).

[69] Mockapetris, P. V. and K. J. Dunlap, ‘‘Development of the Domain Name System,’’ *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, Stanford, CA, pp. 123-133 (August 1988).

[70] Mockapetris, P. V., ‘‘DNS Encoding of Network Names and Other Types,’’ Request for Comments 1101 (April 1989).

[71] Mogul, J. C., ‘‘Representing Information about Files,’’ Ph.D. Thesis, Stanford University, Stanford, CA (March 1986). Available as Stanford Technical Report STAN-CS-86-1103.

[72] Neufeld, G. W., ‘‘Descriptive Names in X.500,’’ *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, Austin, pp. 64-71 (September 1989).

[73] Neuman, B. C., ‘‘The Prospero File System: A Global File System Based on the Virtual System Model,’’ *Computing Systems* **5**(4), pp. 407-432 (1992).

[74] Noll, J. and W. Scacchi, ‘‘Integrating Diverse Information Repositories: A Distributed Hypertext Approach,’’ *Computer* **24**(12), pp. 38-45 (December 1991).

[75] Oppen, D. C. and Y. K. Dalal, ‘‘The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment,’’ *ACM Transactions on Office Information Systems* **1**(3), pp. 230-253 (July 1983).

[76] Ordille, J. J. and B. P. Miller, ‘‘Nomenclator Descriptive Query Optimization in Large X.500 Environments,’’ *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, Zurich, pp. 185-196 (September, 1991).

[77] Ordille, J. J. and B. P. Miller, ‘‘Distributed Active Catalogs and Meta-Data Caching in Descriptive Name Services,’’ *Thirteenth International IEEE Conference on Distributed Computing Systems*, Pittsburgh, pp. 120-129 (May 1993).

[78] Ordille, J. J. and B. P. Miller, ‘‘Database Challenges in Global Information Systems,’’ *ACM SIGMOD International Conference on Management of Data*, Washington, DC, pp. 403-407 (May 1993).

[79] Ozsu, M. Tamer and P. Valduriez, *Principles of Distributed Database Systems*, Prentice Hall, Englewood Cliffs, NJ (1991).

[80] Peterson, L. L., ‘‘A Yellow-Pages Service for a Local-Area Network,’’ *SIGCOMM ’87 Workshop: Frontiers in Computer Communications Technology*, Stowe, Vermont, pp. 235-242 (August 1987).

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

[82] Pike, R., D. Presotto, K. Thompson, and H. Trickey, ‘‘Plan 9 from Bell Labs,’’ *UKUUG Summer 1990 Conference*, London(July 1990).

[83] Press, L., ‘‘The Net: Progress and Opportunity,’’ *Communications of the ACM* **35**(12), pp. 21-25 (December 1992).

[84] Pu, C., A. Leff, and S.–W. F. Chen, ‘‘Heterogeneous and Autonomous Transaction Processing,’’ *Computer* **24**(12), pp. 64-72 (December 1991).

[85] Ritchie, D. and K. Thompson, ‘‘The UNIX Time Sharing System,’’ *Communications of the ACM* **19**(7), pp. 365-375 (July 1974).

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

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

[88] Saltzer, J., ‘‘On the Naming and Binding of Network Destinations,’’ Request for Comments 1498 (August 1993).

[89] Schatz, B. R., ‘‘Telesophy: A System for Manipulating the Knowledge of a Community,’’ *IEEE GLOBECOM ’87*, Tokyo, pp. 30.4.1-30.4.6 (November 1987).

[90] Schroeder, M. D., A. D. Birrell, and R. M. Needham, ‘‘Experience with Grapevine: Growth of a Distributed System,’’ *ACM Transactions on Computer Systems* **2**(1), pp. 3-23 (February 1984).

[91] Schwartz, M. F., J. Zahorjan, and D. Notkin, ‘‘A Name Service for Evolving, Heterogeneous Systems,’’ *Eleventh ACM Symposium on Operating System Principles*, Austin, pp. 52-62 (November, 1987).

[92] Schwartz, M. F., ‘‘The Networked Resource Discovery Project,’’ *IFIP XI World Congress*, San Francisco, pp. 827-832 (August 1989).

[93] Schwartz, M. F. and P. G. Tsirigotis, ‘‘Experience with a Semantically Cognizant Internet White Pages Directory Tool,’’ *Internetworking: Research and Experience* **2**(1), pp. 23-50 (March 1991).

[94] Schwartz, M. F., D. R. Hardy, W. K. Heinzman, and G. C. Hirschowitz, ‘‘Supporting Resource Discovery Among Public Internet Archives Using a Spectrum of Information Quality,’’ *Eleventh International IEEE*

Conference on Distributed Computing Systems, Arlington, Texas, pp. 82-89 (May 1991).

[95] Schwartz, M. F., A. Emtage, B. Kahle, and B. C. Neuman, ''A Comparison of Internet Resource Discovery Approaches,'' *Computing Systems* **5**(4), pp. 461-493 (1992).

[96] Sechrest, S. and M. McClennen, ''Blending Hierarchical and Attribute-Based File Naming,'' *Twelfth International IEEE Conference on Distributed Computing Systems*, Yokohama, Japan, pp. 572-580 (1992).

[97] Sheldon, M. A., A. Duda, R. Weiss, J. W. O'Toole, and D. K. Gifford, ''A Content Routing System for Distributed Information Servers,'' Technical Report MIT/LCS/TR-578, MIT, Cambridge, MA (August 1993).

[98] Sheltzer, A. B., R. Lindell, and G. J. Popek, ''Name Service Locality and Cache Design in a Distributed Operating System,'' *Sixth International IEEE Conference on Distributed Computing Systems*, Cambridge, MA, pp. 515-522 (May 1986).

[99] Sheth, A. P. and J. A. Larson, ''Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases,'' *ACM Computing Surveys* **22**(3), pp. 183-236 (September 1990).

[100] Shirriff, K. W. and J. K. Ousterhout, ''A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System,'' *USENIX*, San Francisco, pp. 315-330 (Winter 1992).

[101] Shoch, J. F., ''Inter-Network Naming, Addressing, and Routing,'' *IEEE COMPCON '78*, Washington, DC, pp. 72-79 (September 1978).

[102] Smetaniuk, B., ''Distributed Operation of the X.500 Directory,'' *Computer Networks and ISDN Systems* **21**, pp. 17-40 (1991).

[103] Sollins, K. R. and D. D. Clark, ''Distributed Name Management,'' *IFIP TC 6/WG 6.5 Working Conference on Message Handling Systems*, Munich, pp. 97-115 (April 1987).

[104] Sollins, K. R., ''Plan for Internet Directory Services,'' Request for Comments 1107 (July 1989).

[105] Sollins, K. R., ''Supporting the Information Mesh,'' *Workshop on Workstation Operating Systems*, Miami(April, 1992).

[106] Soparkar, N., H. F. Korth, and A. Silberschatz, ''Failure–Resilient Transaction Management in Multidatabases,'' *Computer* **24**(12), pp. 28-36 (December 1991).

[107] Steiner, J. G., B. C. Neuman, and J. I. Schiller, ''Kerberos: An Authentication Service for Open Network Systems,'' *Usenix Conference*, Dallas, pp. 191-202 (Winter 1988).

[108] Stonebraker, M., ''The Design and Implementation of Distributed INGRES,'' pp. 187-196 in *The INGRES Papers*, ed. M. Stonebraker, Addison-Wesley Publishing, Menlo Park, CA (1986).

[109] Terry, D. B., ''Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments,'' Ph.D. Thesis, University of California at Berkeley (February 1985). Available as Technical Report CSL-85-1, Xerox Palo Alto Research Center.

[110] Terry, D. B., ‘‘Structure-free Name Management for Evolving Distributed Systems,’’ *Sixth International IEEE Conference on Distributed Computing Systems*, Cambridge, MA, pp. 502-508 (May 1986).

[111] Terry, D. B., ‘‘Caching Hints in Distributed Systems,’’ *IEEE Transactions on Software Engineering* **13**(1), pp. 48-54 (January 1987).

[112] Theimer, M., L.-F. Cabrera, and J. Wyllie, ‘‘QuickSilver Support for Access to Data in Large, Geographically Dispersed Systems,’’ *Ninth International IEEE Conference on Distributed Computing Systems*, Newport Beach, pp. 28-35 (May 1989).

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

[114] Walker, B., G. Popek, B. English, C. Kline, and G. Thiel, ‘‘The LOCUS Distributed Operating System,’’ *Ninth ACM Symposium on Operating Systems Principles*, Bretton Woods, NH, pp. 49-70 (October 1983).

[115] Watson, R., ‘‘Identifiers (Naming) in Distributed Systems,’’ pp. 191-210 in *Distributed Systems – Architecture and Implementation*, ed. H. J. Siegert, Springer-Verlag, Berlin (1986).

[116] Weider, C., J. Fullton, and S. Spero, ‘‘Architecture of the WHOIS++ Index Service,’’ Internet Draft, Merit Network, Inc. (March 1993).

[117] Welch, B. 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).

[118] White, J. E., ‘‘A User-Friendly Naming Convention For Use in Communication Networks,’’ *IFIP WG 6.5 Working Conference on Computer-Based Message Services*, Nottingham, England, pp. 39-57 (May 1984).

[119] Wiederhold, G., ‘‘Mediators in the Architecture of Future Information Systems,’’ *Computer* **25**(3), pp. 50-62 (March 1992).

[120] Williams, R., D. Daniels, L. Haas, G. Lapis, B. G. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost, ‘‘R*: An Overview of the Architecture,’’ pp. 196-218 in *Readings in Database Systems*, ed. M. Stonebraker, Morgan Kaufmann Publishers, Palo Alto, CA (1988).

TABLE OF CONTENTS