Distributed Active Catalogs and Meta-Data Caching in Descriptive Name Services

Joann J. Ordille joann@cs.wisc.edu
Barton P. Miller bart@cs.wisc.edu

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

Abstract

Today's global internetworks challenge the ability of name services and other information services to locate data quickly. We introduce a distributed active catalog and meta-data caching for optimizing queries in this environment. Our 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. 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. When searching for query responses, our techniques contact only the small percentage of the data repositories with actual responses, resulting in search times of a few seconds. We implemented a distributed active catalog and meta-data caching in a prototype descriptive name service called ''Nomenclator.'' We present performance results for Nomenclator in a search space of 1000 data repositories.

1. Introduction

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 information the users need. System capacity increases if we avoid unnecessary communication with data repositories. We improve the performance of descriptive (i.e. relational) name services in this highly distributed environment by providing a single framework for constraining the search space and reducing processing overhead. We introduce *distributed 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. We introduce *general meta-data caching* to reduce processing overhead, and integrate our new meta-data techniques with existing data caching techniques. 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.

This work was supported in part by an AT&T Ph.D. Scholarship, National Science Foundation grants CCR-8815928 and CCR-9100968, Office of Naval Research grant N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.

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 [1] 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. The active catalog uses meta-data descriptions, called *referrals*, to specify the conditions for using catalog functions. Graphs of referrals allow us to select the right catalog function for our needs and reap the benefits of multiple catalog functions in processing one query.

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 contact only the small percentage of the data repositories with actual responses, resulting in search times of a few seconds. Even a request to search the global name space for a person with a popular name can be answered in seconds.

Distributed active catalogs and meta-data caching are currently used in a prototype descriptive name service called Nomenclator [10]. Nomenclator answers selection and projection queries on relations that span heterogeneous name services in the global internetwork. Like the Domain Name System [8], Nomenclator currently uses timestamps to identify and replace potentially stale data and meta-data in its cache. In the naming environment, cached information is not required to be consistent, but to converge eventually to the new information after an update [2, 8]. A delay in seeing a change is only an inconvenience, and it is a rare inconvenience because data and meta-data change infrequently. Users see either an old or a new version of each tuple; collections of tuples have no inter-dependencies that would require the consistency of multiple tuple transactions.

The following sections describe our research in more detail. Section 2 provides an overview of the Nomenclator System Architecture. Section 3 describes referrals and explains how referrals form a graph that guides query processing. Section 4 describes catalog functions and the techniques for generating referrals dynamically without storing the entire referral graph. We extend the referral format in Section 5 to provide additional opportunities for faster, user-friendly query processing. Section 6 gives experimental results that show our techniques improve performance for a wide range of data distribution patterns and response sizes in a search space of 1000 data repositories. Finally, Section 7 describes related work, and Section 8 provides a summary.

2. Nomenclator

The distributed active catalog promotes sharing of information in environments that require distributed information control like today's global name spaces. The Nomenclator name service implements the distributed active catalog using a distributed catalog service and a query resolver (see Figure 1). The *distributed catalog service* supplies meta-data for each relation, including referrals, catalog function definitions, and the names of attributes. It provides an opportunity for the owners of data to advertise their information on the network. While the distributed catalog service supplies meta-data for well-known name services, like X.500 [4] and the Domain Name System, it also encourages the owners of other data to provide instruction on how to find their data. In the simpliest case, the owners tell the distributed catalog service where their data is located. More generally, they can provide catalog functions to constrain searches of the owners' data repositories, and we provide tools for generating these catalog functions. Organizations with proprietary information or that build value-added indices for information in the network can preserve their privacy by providing catalog function services via remote procedure calls.

The *query resolver* accepts and answers queries from users. It is a data driven query processing engine fueled by referrals. The resolver imports referrals to

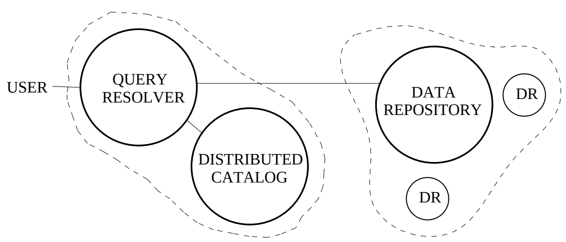


Figure 1.
Nomenclator System Architecture.

catalog functions from the distributed catalog service. These functions generate additional referrals that the resolver can cache and re-use as appropriate. Typically, one resolver process serves several users on a local area network, so users can benefit from a larger resolver cache.

3. Referrals

Referrals are a general mechanism for describing distributed indexing structures. They direct query processing by stating the conditions for using *access functions*. The first type of access functions, catalog functions, constrain the search space for a query. They direct the search to more selective indices by returning referrals to other catalog functions. They constrain the search to specific data repositories by returning referrals to the other type of access functions, called *data access functions*. Data access functions encapsulate the heterogeneous access methods in the name space by mapping queries to the access operations of a data repository. They return tuples that answer the query.

Each referral contains a template and a list of references to access functions (see Figure 2). The template is a selection predicate that describes the scope of the the access functions. Our system follows the following rule:

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

For example, the first referral in Figure 2 covers the queries in Figure 3.

Referrals can describe the partitioning criteria of a relation, and also describe more complex indexing structures. For example, the `People` relation is partitioned by organization, and we describe this partitioning criteria by a series of referrals like the second referral in Figure 2. This referral contains a data access function depicted with the data repository that it uses to answer queries, and it also covers the first query in Figure 3. Many distributed database systems, like Distributed INGRES [16], use an approach similar to ours for describing physical partitions of a relation, but the distributed active catalog also does

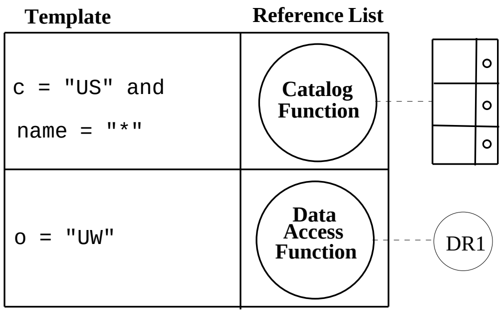


Figure 2.
Sample Referrals for the People *Relation.*

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

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

Figure 3.
Sample Queries about People in Organizations in the United States.

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 2 describes one such index. It contains a catalog function depicted with the list of referrals it returns. This catalog function returns referrals that describe the distribution of a particular surname on data repositories in the United States. The wildcard in name indicates that the catalog function can constrain the search space more effectively if name 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 [7], Domain Name System [8], and R* [17], 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 view of the relation, like the catalog function in Figure 2.

Referrals form a generalization/specialization graph for a relation called a *referral graph*. Referral graphs integrate the different catalog functions in our system, and supply 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: $s \subset g$. Referral s is a subset of referral g if the template for s is a subset of the template for g. s is considered a *more specific* referral than g; g is considered a *more general* referral than s.

Part of the referral graph for the People relation is shown in Figure 4. 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. The arcs in the graph indicate the path from a general referral to a more specific referral. Notice that referrals r1, r2, and r3 are ordered from general to specific, but that r1 and r4 (and r3 and r6) are not ordered by the graph. The direction of the arcs also indicates the direction in which the search space is constrained. The first query in Figure 3 is covered by referral r3 and also by referral r1, but it is answered using r3, the more constrained (and faster) referral.

The resolver query processing algorithm 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. For example, a query about a person in the Computer Sciences Department at the University of Wisconsin is constrained by referrals r2 and r4 in Figure 4. The intersection of these 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. 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, and a generalization of data indices for a large internet environment. 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 sources that are obtained by the query

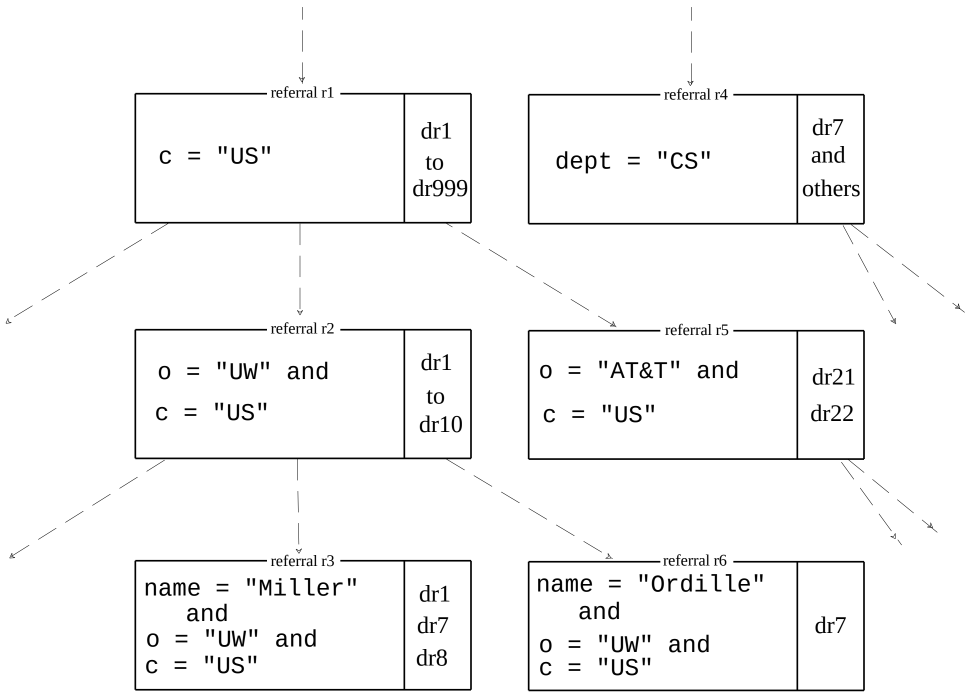


Figure 4.
Sample Partial Referral Graph.

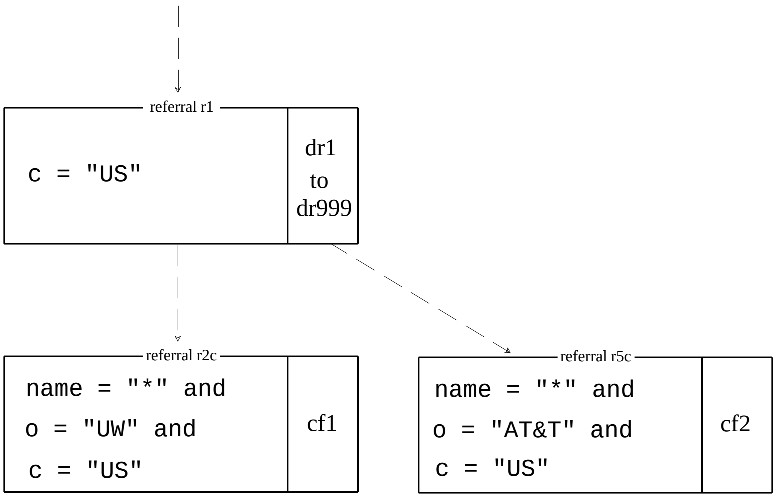


Figure 5.
Catalog Functions Encapsulate Parts of a Referral Graph.

resolver from the distributed active catalog. The resolver dynamically compiles and loads them into its address space using an approach similar to CLAM [3].

Catalog functions encapsulate portions of the referral graph. For example, catalog function `cf1` in Figure 5 uses the attribute `name` to direct queries to data repositories at the University of Wisconsin. It encapsulates referral `r2` (from Figure 4) and its more specific children. This catalog function can return referrals `r2`, `r3` or `r6` as appropriate. 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 template `t` for a catalog function `cf`, and a query $q \subseteq t$, the result of using `cf` to process `q`, `cf(q)`, is a referral with template t' such that $q \subseteq t'$ and $t' \subset t$.

Catalog functions can also encapsulate other catalog functions by calling them. For example, we can replace the entire graph in Figure 5 with the referral in Figure 6. The catalog function `cf3` calls `cf1` and `cf2`, and returns the union of their results. 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.

5. 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 3 using `cf3`, we receive a referral with the template `name = "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 find previously cached subquery answers (in this case, a query for "Ordille" in a particular organization) and 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 7(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 `r1` in Figure 4 to the the referral in Figure 7(b). The resulting referral is the union of leaves of the referral graph; its revised templates and access functions are the templates and access functions of the leaves.

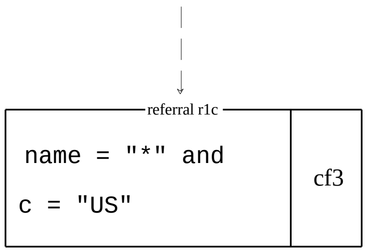


Figure 6.
A Catalog Function Encapsulates Figure 4.

The construction of general referrals with template `t` and revised templates `rt1`, `rt2`, ..., `rt3` follows two rules. The first rule is the following:

> *Referral Coverage Rule:* $t \subseteq rt1 \cup rt2 \ldots \cup rtn$.

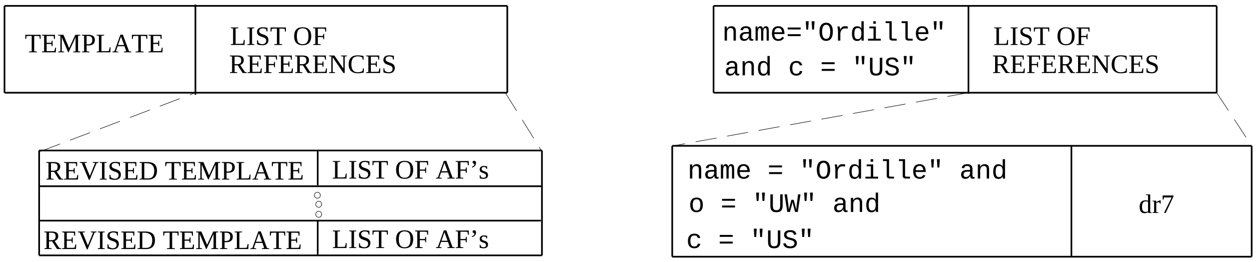
This rule, like the Query Coverage Rule, is required for correctness. Catalog functions forming the union of referrals must know that the union covers the scope of the returned template. The catalog function `cf3` can only return the referral in Figure 7 (b), because it has contacted every organization in the United States and found only one place where "Ordille" is listed. The second rule is the

> *Referral Constrained Search Rule:* $t \supseteq rt1 \cup rt2 \ldots \cup rtn$.

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.

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 inconsistent, the contents of the data repository contradict the query and the data repository is not contacted. We plan 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 selects attribute values from the list to constrain the query further.

Referrals and the four simple rules summarized in Table 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



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

Figure 7.
Referrals.

Rule Name	**Rule Summary**
Query Coverage Rule	**If** $q \subseteq t$**, then use referral.**
Catalog Function Constrained Search Rule	**If** $q \subseteq t$ **and** $cf(q)$ **returns** t'**, then** $q \subseteq t'$ **and** $t' \subset t$.
Referral Coverage Rule	$t \subseteq rt1 \cup rt2 \ldots \cup rtn$.
Referral Constrained Search Rule	$t \supseteq rt1 \cup rt2 \ldots \cup rtn$.

Table 1.
Meta-Data Rules.

(for a query q, a referral with template t', and a referral with template t, catalog function cf, and revised templates rt1, rt2, ... rtn.)

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[6] to cache and re-use the 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.

6. Experiments

Three issues are important in evaluating our query processing framework. First, we must determine whether we 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 our query processing framework given the existence of constrained search spaces. Can we find information in the global name space quickly? Third, we must analyze the scaling behavior of our framework for query workloads from multiple users. How well will our query processing scale to millions of users?

Our previous work shows that it is possible to constrain queries in a real environment, the X.500 name space [10]. In that study, for example, the active catalog constrains the attribute `surname` with value "Miller" to only 32% of the X.500 data repositories in the United States. `Surname` is an attribute users are likely to know when searching for information about people. Moreover, even a common surname like "Miller" greatly reduces the number of data repositories searched. When more unusual surnames are used (like "Ordille") or additional attributes are specified, we can do even better in constraining the search space.

In this 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 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 names services like the Knowbot Information Service [5].

These experiments do not measure the effects of parallel query processing at data repositories or the interactions of multiple catalog functions. The costs of dynamically loading access functions and validating caches are not analyzed in this study, because these costs must be weighed against the benefits to a stream of queries. Moreover, dynamic loading costs are typically low [3], 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.

These first two studies investigated how Nomenclator reduces the impact of the individual user on the query processing environment. We are currently studying how Nomenclator manages queries from many users to achieve scalable system performance. Our continuing research will determine the effectiveness of caches for streams of queries. We are also studying the tradeoff between the costs of maintaining access functions, and the aggregate savings (for large workloads) from constraining the search space and caching.

6.1. 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[1] 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.

6.2. 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 referral cache, and by Nomenclator with a warm referral cache. When Nomenclator had a cold 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.

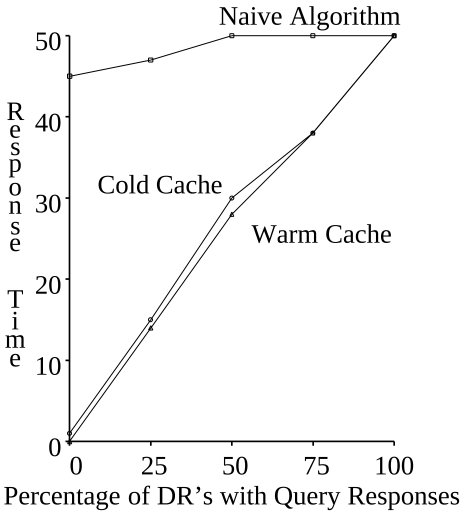


Figure 8.
Response Time Results for 1 Byte Response Tuples.

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

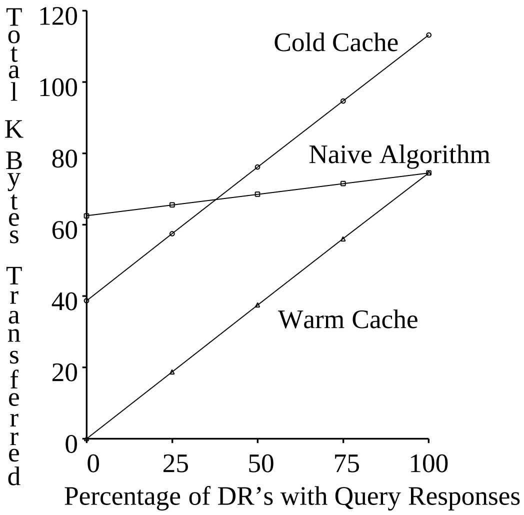


Figure 9.
System Load Results for 1 Byte Response Tuples.

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

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 8 reports the response time measurements for the queries where the data repositories returned one byte tuple responses, and Figure 9 reports the number of bytes transferred by those queries. The number of bytes 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 10 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.

6.3. Discussion

Our experiments show that our techniques to eliminate data repositories from the search space can dramatically improve response time. As we anticipated, Figures 8 and 10 report a linear relationship between the number of data repositories contacted and the response time. Our techniques successfully eliminate unnecessary work from the query processing without introducing new bottlenecks. Both graphs show significant response time improvements, because 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 10 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 10 shows that even large name service queries will be answered quickly. When 30 percent or less of the data repositories contain responses, both Figures 8 and 10 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 9 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,

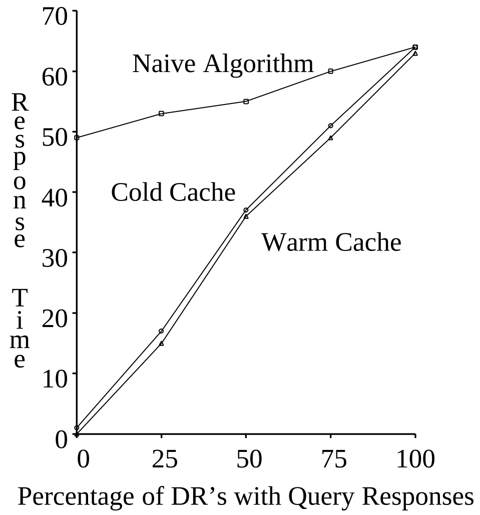


Figure 10.

Response Time Results for 1000 Byte Response Tuples.

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

Nomenclator exceeds the load of the naive algorithm when more than 35 percent of data repositories 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 9 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 8 and 10 show less dramatic performance gains from meta-data caching for single users, because latency was low in our test environment. This improvement will be much 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 greater percentage of the processing time in this operating range.

7. Related work

Previous descriptive name services limit the extent of data distribution or types of descriptive queries to attain performance. Profile [12] processes descriptive queries by contacting every entry in a path of data repositories. The path is specified by the user or discovered in the data during query processing. Performance is limited by the length of the chain of servers contacted. Unlike Nomenclator, Profile does not use caching or information about data distribution patterns to improve performance. Profile allows users to specify *preferences* about the importance and use of attributes in the query resolution search. Some of these preferences increase performance by constraining the scope of the search while others specify ways to interpret attributes in the name space. Nomenclator differs from Profile in that owners of data, not users, provide information that guides the search.

X.500 [4] provides a descriptive query called `SEARCH`. This query is limited in performance, because it exhaustively searches subtrees in the X.500 name space. Neufeld [9] improves the performance of `SEARCH` for a subset of queries by augmenting the X.500 partitioning criteria with *registered attribute values*. Registering a value guarantees that the attribute will have only that value in a subtree of the name space. Using registered values to constrain the search is similar in method and utility to improving query performance by using the partitioning criteria of a relation. We prefer a more general approach that uses information about data distribution patterns to improve performance and does not require owners of data to constrain the values of attributes in other organizations.

The Networked Resource Discovery Project [13, 14] provides an architecture for locating a few instances of a type of resource when the resource type is prevalent in the network. It multicasts queries to a probabilisticly chosen subset of the available data repositories. Successive queries do not return the same answer, and queries may fail even when data satisfying the query is present in the system.

Multidatabase and federated database systems [15] typically follow the lead of distributed database systems [11] in achieving selection predicate performance. These systems limit their opportunities for optimization to using the partitioning criteria of a relation to constrain the search space. While this approach is useful in small systems and on local area networks, it does not scale to systems with thousands of data repositories. While multidatabase and federated systems translate and forward each query to all their component systems, Nomenclator only performs these operations when the destination has data that may be relevant to the query response.

8. Summary

Distributed active catalogs and meta-data caching are new techniques for improving selection predicate performance in very large, distributed environments. The active catalog is a distributed facility that constrains queries to those data repositories where query answers are likely to exist. Meta-data caching keeps frequently used components of the active catalog available locally. It stores the results of constraining the search space, so they can be re-used without additional costs. Referral graphs provide a single framework for using the distributed active catalog, meta-data caching and data caching in query processing. Our experiments indicate that these techniques improve response time and reduce system load for a wide range of data distribution patterns. In our typical operating range, queries that take minutes using current strategies can be answered in a few seconds using our techniques. Our techniques are appropriate for environments with loose consistency constraints and we hope to extend them to systems with stronger consistency constraints.

9. Acknowledgments

We are grateful to James Elliott for his work on the Nomenclator parser and user interface, and to Cheryl Thompson for her work on an X Window interface to Nomenclator.

10. References

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

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

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

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

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

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

[7] D. K. Gifford, 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, Washington, pp. 161-169 (December 1985).

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

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

[10] J. J. Ordille 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).

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

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

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

[14] M. F. Schwartz, ‘‘A Scalable, Non-Hierarchical Resource Discovery Mechanism Based on Probablistic Protocols,’’ Technical Report CU-CS-474-90, University of Colorado, Boulder, Colorado (June 1990).

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

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

[17] R. Williams, 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).