

Distributed Evaluation of Network Directory Queries

Sihem Amer-Yahia Divesh Srivastava
AT&T Labs–Research
{sihem, divesh}@research.att.com

Dan Suciu
University of Washington
suciu@cs.washington.edu

Abstract

This paper describes novel efficient techniques for the distributed evaluation of hierarchical aggregate selection queries over LDAP directory data, distributed across multiple autonomous directory servers. Such queries are useful for emerging applications like the Directory Enabled Networks initiative. Our techniques follow the LDAP approach of distributed query evaluation by referrals, where each relevant server computes answers locally, and the LDAP client coordinates between directory servers. We make a conceptual separation between the identification of relevant servers and the distributed computation of answers. We focus on the challenging task of generating an efficient plan for evaluating hierarchical aggregate selection queries, which involves correlating directory entries across multiple servers. The key features of our plan are: (i) the network traffic consists of query answers, and auxiliary messages that depend only on the number of servers and the size of the query (not on the data size), (ii) the coordination effort at the client is independent of the data size, and (iii) potentially expensive server-to-server communication and coordination is avoided. We complement our analysis with experiments that show the robustness and scalability of our techniques for highly distributed directory query processing.

Index: LDAP directories, hierarchical aggregate selection, distributed query evaluation, aggregate value cache, topology identification.

1 Introduction

LDAP directories [5, 6, 7] have rapidly emerged as an essential component of the network infrastructure, and are being used to store a wide variety of information for enabling integration among applications and services on the network. These include user profiles and address books for messaging applications, network security policies, and network management and customer policies for the recent DEN (Directory Enabled Networks) initiative [1, 11].

Directory data is often highly distributed on a collection of autonomous directory servers, while allowing for logical centralization and conceptual unity using a hierarchical naming convention, in a way that is not well supported by relational databases. The conceptually unified nature of distributed directories encourages users to pose queries without having to be aware of the location of individual entries. It is the task of the directory servers and the directory client to evaluate these queries in a distributed fashion. In the approach preferred by LDAP implementations, called *distributed evaluation by referrals*, each relevant directory server computes local answers to the distributed query, and the directory client coordinates interaction with the servers; distributed query evaluation does not involve any server-to-server communication and coordination. The autonomous nature of the servers that form the distributed directory, especially in the wide-area Internet, makes this approach the distributed LDAP query evaluation strategy of choice.

LDAP directories currently support only boolean selection queries, which do not match the needs of emerging directory applications. For example, in a DEN security application, a natural query is to identify the highest priority firewall management policy in the directory that matches a given profile. Doing so using LDAP queries results in both a large amount

of network traffic, and in a computationally intensive client application. Jagadish et al. [7] identified this problem and proposed *hierarchical aggregate selection* queries for this new generation of directory applications. They presented efficient algorithms (with linear CPU and I/O complexities) for centralized evaluation of such queries. Given the ubiquitous nature of distributed directory data, efficient distributed evaluation of the hierarchical aggregate selection queries of [7] is important. This is the subject of this paper.

Our novel techniques are based on the LDAP approach of distributed evaluation by referrals, and make a conceptual separation between (i) the identification of servers relevant to a query, and (ii) the distributed computation of query answers present at each of the relevant servers. This distinction enables us to focus on the challenging task of generating an efficient plan for the distributed evaluation of hierarchical aggregate selection queries, which involves correlating entries across multiple servers; in particular, an answer to a hierarchical aggregate selection query may be present in one server, while verifying whether it is in fact an answer may involve accessing multiple other servers. Distributed evaluation of a (boolean) LDAP query involves no such correlations and can be evaluated independently at each of the relevant servers. The key features of our distributed query evaluation plan are:

- The total network data traffic consists of the query answer, along with sub-queries and aggregate values whose sizes depend only on the number of directory servers and the size of the original query, but not on the total size of the directory data.
- The total coordination effort at the directory client is independent of the size of the directory data.

We complement our analysis with experimental results, based on a real implementation

that modified the open source OpenLDAP directory server [2] and enhanced the LDAP client library, that show the robustness and scalability of our algorithms for highly distributed directory query processing.

- By separating the identification of relevant servers from the computation of query answers, the performance of distributed query evaluation is *insensitive* to the actual topology of the directory servers.
- By using an aggregate value cache at the client, the total cost of distributed directory query evaluation scales linearly with the number of directory servers.

The rest of this paper is organized as follows. We start by a discussion of the related work in Section 1.1. Then, we present the directory data model, and the basics of distributed directories in Section 2. Section 3 contains an overview of directory queries, and issues that arise in their distributed evaluation. The intricate details of generating an efficient distributed plan for evaluating hierarchical aggregate selection queries are discussed in Section 4. Finally, experimental results quantifying the benefits of our techniques are presented in Section 5.

1.1 Related Work

There has been a lot of work in the evaluation and optimization of relational and object-oriented queries over distributed data (see, e.g., Özsu and Valduriez [10] and Kossman [8]). Since directory queries select entries that satisfy a (possibly complex) condition, most relevant to us is the work in the distributed evaluation of SQL join and semijoin queries.

There are three key differences between this body of work and ours. First, there are no data-independent bounds on the amount of non query answer data traffic generated for SQL queries. It is possible, in principle, that this traffic can substantially dominate the query answer data traffic. The main reason for this difference is that relational joins and semijoins in SQL queries are *value-based*, whereas the correlations inherent in the hierarchical aggregate selection directory queries are based on the hierarchical directory *structure*. Our techniques make critical use of this aspect of directory queries to obtain very efficient query evaluation plans, even for highly distributed directories.

A second difference is that the distributed evaluation of SQL queries involves a substantial, data-dependent, tradeoff between communication and computation. The use of semijoin-style optimizations in distributed SQL query evaluation, for example, attempts to reduce communication cost at the expense of increased per-server computation cost [3, 9]. Our techniques for the distributed evaluation of hierarchical aggregate selection queries, however, allow per-server computation costs to be optimized *independently* of communication costs, in a data-independent manner.

Third, the approaches used for the distributed evaluation of SQL queries typically involve server-to-server communication to correlate data between servers. This is appropriate when the servers belong to the same organization, connected by high bandwidth, dedicated links. In contrast, our work is based on the LDAP approach of distributed query evaluation by referrals, where the directory client is responsible for coordinating interaction with the multiple servers, and inter-server communication and coordination is avoided during query processing; this is more appropriate for servers on a wide-area network, such as the Internet.

2 Background

2.1 Directory Data Model

The directory data model (see, e.g., [7]) defines a set of classes and attributes. An instance of a class is called an entry. Each entry r has a distinguished name $dn(r)$, “holds” information in the form of a set of (attribute, value) pairs $val(r)$, and “belongs to” a non-empty set of classes $class(r)$. An entry may have multiple values for an attribute; in particular, the classes that entry r belongs to are precisely the (multiple) values of r ’s `objectClass` attribute.

Distinguished Names & Hierarchical Namespaces: The distinguished name, $dn(r)$, of a directory entry r is a sequence s_1, \dots, s_n of (attribute, value) pairs, and serves to uniquely identify the entry; the first component, $s_1 \in val(r)$, is called the *relative distinguished name* of r , denoted by $rdn(r)$. See Figure 1 for examples. Distinguished names naturally induce a *hierarchical namespace* among directory entries. We say that: (a) entry r is a *parent* of entry r' if $dn(r') = rdn(r'), dn(r)$; entry r' is said to be a *child* of entry r . (b) entry r is an *ancestor* of entry r' if there exist (attribute, value) pairs s_1, \dots, s_m s.t. $dn(r') = s_1, \dots, s_m, dn(r)$; entry r' is then said to be a *descendant* of r . We refer to this hierarchical organization as the *directory information forest* (DIF).

Next, we present a running example, to be used throughout the paper, for supporting network firewall policies, an important directory enabled network (DEN) application.

Example 2.1 [Network Firewall Policies]

A DEN represents, in a directory, profiles of network users, applications and services, as well as policies for the overall management of the network (see, e.g., [1, 11]).

packets that the policy applies to, and the policy's *action* specifies the treatment of packets so identified. A policy conflict may occur when two or more policies in the directory specify conflicting actions for the same packet. A simple conflict resolution approach is based on a *priority* attribute associated with each policy; in case of conflicts, the applicable policy with the highest priority is applied.

Figure 1(a) depicts sample directory data useful for supporting a firewall policy within a subnet of an Internet Service Provider, based on the schema descriptions in [4]. Four classes are defined: `SLAPolicy` (each instance entry represents a distinct policy), `trafficProfile` (each instance entry represents a network packet profile), `policyValidityPeriod` (each instance entry represents a temporal applicability of the policy), and `SLADSAction` (each instance entry represents an action to be performed). For example, the policy with *dn* `SLAPolicyName=dso, ou=firewallPolicies, dc=subnet10, dc=ISP, dc=com` is relevant to data traffic whose IP source address matches `204.178.16.*`, during weekends in 1999 and 2000. All packets that match this profile are denied permission by the firewall into the subnet. Similar policies may additionally be specified for this subnet, for other subnets in the network, and for the network as a whole. ■

2.2 Distributing Directory Data

The hierarchical namespace induced by the distinguished names of entries allows a natural mechanism for *distribution*, in which entries are partitioned across multiple directory servers. Each directory server manages the entries in a single directory partition, and is hence identified with that partition. This distribution is hidden from the user who is presented with

the directory information forest (DIF) as a conceptually unified view of the directory data.

Definition 2.1 [Directory Partition] A *directory partition* of a set of directory entries R is a triple (N, E, π) , such that: (i) N denotes a set of partitions, $E \subset N \times N$, and the graph (N, E) must be a forest; (ii) the function $\pi : R \rightarrow N$ associates with each entry $r \in R$ its partition in N ; (iii) $\forall n \in N, \exists r \in R$ s.t. $\pi(r) = n \wedge (\forall r' \in R$ if $\pi(r') = n$ then r' is a (non-proper) descendant of r); this common ancestor r is referred to as the *partition root* of n ; and (iv) $\forall r, r' \in R$, if r is an ancestor of r' , then the edge $(\pi(r), \pi(r'))$ must appear in the reflexive, transitive closure of (N, E) . ■

The top part of Figure 1(b) illustrates how the entries in a conceptually unified DIF can be distributed among multiple directory partitions (indicated by dashed rectangles). The partitions are themselves hierarchically related, and the relationships between them are achieved via “superior” and “subordinate” knowledge references (indicated by triangles and circles in the bottom part of the figure) and are modeled as entries belonging to the class `referral` in the directory servers that manage the partitions. Referral entries contain the LDAP URL of the higher or the lower directory server (including name, port number and *dn* of its partition root) as values of the `ref` attribute (indicated by dashed arrows in the bottom part of Figure 1(b)).

The hierarchical directory namespace typically corresponds to administrative responsibilities for portions of the namespace, and may reflect, e.g., political, geographic, and/or organizational boundaries. This is very similar to the way the Domain Name System (DNS) operates, which allows maintenance of its (hierarchical) namespace in a distributed fashion, and provides rapid lookups in the namespace.

3 Distributed Query Evaluation

This paper is concerned with distributed evaluation of directory queries over the hierarchical structure of directory servers. To illustrate the complexity of the problem, and hint at our solution, for hierarchical aggregate selection queries, we focus first on boolean LDAP queries.

We assume a distributed directory with four partitions, each managed by a separate server (see Figure 1(b)). Our discussion focuses on two complexity measures that contribute to the response time observed by the user: the total number of messages exchanged, and the total amount of data exchanged in these messages.

3.1 Boolean LDAP Query Evaluation

LDAP Queries: An LDAP query is like an SQL selection query and is specified by:

```
base-entry-DN ? scope ? boolean-filter (1)
```

The *base entry DN* of a query identifies the root of a subtree in the DIF. The *scope* indicates whether the filter has to be evaluated only at the base entry (**base**), only at the children of the base entry (**one**), or down from the base entry to all its descendants (**sub**). The boolean filter [5, 12] compares individual attributes with values. Filters can be combined using the standard boolean operators: **and** (&), **or** (|), **not** (!). A query returns one or more directory entries along with their attributes.¹

Directory queries (in our motivating application) are issued by policy enforcement entities such as hosts, firewalls, and proxy servers. When (data or signaling) packets of a certain

¹LDAP queries also permit projections, but these are irrelevant for our discussion.

type are encountered, e.g., an RSVP message, or a TCP connect request, the enforcement entity provides the policy server with: (a) the attributes of the packet header (source address and port, destination address and port, etc.), and (b) the current time, as the profile to be matched against the directory. The policy server then interacts with the directory to identify and return the actions that are specified by the highest priority policies that match the given profile. The policy enforcement entity then applies these actions to all packets in that flow, without having to query the directory repeatedly for each packet encountered in the flow.

Commercial LDAP directory servers are tuned to support fast querying, and the query response times of these servers are adequate for rapid enforcement of network firewall policies. Let us consider the evaluation of the following LDAP query Q that asks for policies:

```
dc=subnet9, dc=ISP, dc=com ? sub ? (objectClass=SLAPolicy)
```

and assume that the base of Q is managed by server $S2$.

Distributed Evaluation by Referrals: LDAP's preferred distributed evaluation mode is called *by referrals*, which works as follows. Each client application knows the name of a directory server: assume in our example this is server $S4$. The client first sends the query Q to $S4$, which replies "I don't manage the base `dc=subnet9, dc=ISP, dc=com`, try (my parent) server $S1$ ". Next, the client sends query Q to $S1$, which replies: "I don't manage this base, try (my child) server $S2$ ". Now the client sends query Q to $S2$, which finally holds the base, and replies with a set of entries satisfying query Q . Since the query scope is `sub`, $S2$ also returns a subordinate referral to server $S3$, which the client contacts by sending Q (with its base entry DN modified to be the partition root of $S3$). The client's answer consists of the union of the sets of entries returned by $S2$ and $S3$.

It is easy to see that the total number of rounds of messages exchanged between the client and the servers in the worst case is almost twice the number of servers. The total amount of data exchanged in these messages is the size of the query's answer plus almost twice the number of servers. This leads to the following result:

Proposition 3.1 *Consider a boolean LDAP query Q , evaluated by referrals against a distributed directory with n partitions (servers). Then, the total number of rounds of messages exchanged between the directory client and the servers is $O(n)$. The total amount of data exchanged in these messages is $O(|Ans(Q)| + n)$, where $Ans(Q)$ is the answer to Q . ■*

For reasons of space, we do not discuss the distributed evaluation strategy that involves inter-server communication, *by chaining*, except to mention that it is not as efficient in general, and is rarely supported in commercial LDAP implementations.

3.2 Hierarchical Aggregate Selection

Let us consider our motivating application again. It is easy to see that one cannot efficiently identify the highest priority `SLAPolicy` entry in the directory that matches a given profile, using a single LDAP query. This problem was identified by Jagadish et al. [7], who then proposed hierarchical aggregate selection queries, as an extension of LDAP queries, to resolve the problem. Here, we present the hierarchical aggregate selection queries of [7], using a slightly different syntax to enable easier understanding of the distributed plan generation.

We first discuss aggregate value expressions that result in numeric values, and then discuss hierarchical aggregate selection queries that (like LDAP queries) return directory entries.

Aggregate Value Expression	Expression Result
$(\text{sum } selQ \ valueExpr)$	sum of $valueExpr$ values for each entry satisfying $selQ$
$(\text{min } selQ \ valueExpr)$	minimum of $valueExpr$ values for each entry satisfying $selQ$
$(\text{max } selQ \ valueExpr)$	maximum of $valueExpr$ values for each entry satisfying $selQ$

Figure 2: Meaning of Aggregate Value Expressions

Selection Query	Query Result
$(d \ selQ \ aggCond)$	entries satisfying query $selQ$ whose set of descendants satisfy $aggCond$
$(c \ selQ \ aggCond)$	entries satisfying query $selQ$ whose set of children satisfy $aggCond$
$(a \ selQ \ aggCond)$	entries satisfying query $selQ$ whose set of ancestors satisfy $aggCond$
$(p \ selQ \ aggCond)$	entries satisfying query $selQ$ whose parent satisfies $aggCond$
where $aggCond$ has the form $(aggExp \ oprel \ valueExpr)$, $oprel \in \{>, <, \geq, \leq, =, \neq\}$	

Figure 3: Meaning of Hierarchical Aggregate Selection Queries

Aggregate Value Expressions: Aggregate value expressions are summarized in Figure 2. $valueExpr$ is an arithmetic expression involving attribute names, constants, and arithmetic operators, e.g., $PVDayOfWeek$ and $(PVStopTime - PVStartTime)$. Our examples typically use aggregate value expressions involving `count`; note that even though `count` is not a primitive aggregate function, it can be expressed as: $(\text{count } Q) = (\text{sum } Q \ 1)$.

Hierarchical Aggregate Selection Queries: Aggregate value expressions can occur in a query filter (see Equation 1, in Section 3.1), wherever a number is expected. They can also be used to construct more complex *hierarchical aggregate selection queries* [7]. Such a query retrieves entries based on some condition involving an aggregate value. Figure 3 shows all hierarchical aggregate selection queries. For example, the simple query below retrieves

policies having multiple actions specified:

```
(d (dc=ISP, dc=com ? sub ? objectClass=SLAPolicy)
  ((count (dc=ISP, dc=com ? sub ? objectClass=SLADSAction)) ≥ 2))
```

There are four *hierarchical operators*: *d* (descendant), *c* (child), *a* (ancestor), and *p* (parent). We describe now the semantics of a hierarchical aggregate selection query with the *d* operator, $(d \text{ sel}Q \text{ aggCond})$; the others being similar. First *selQ* is evaluated, and results in a set of entries $\{e_1, e_2, \dots\}$. For each entry e_i the aggregate condition *aggCond* is evaluated over its set of descendants: i.e., *aggExp* is evaluated only over the set of e_i 's descendants, resulting in a number, which is compared (using *oprel*) to *valueExpr*. The latter, *valueExpr*, is an arithmetic expression involving attributes from e_i and/or constants. If the comparison evaluates to true, then e_i is included in the result of $(d \text{ sel}Q \text{ aggCond})$.

3.3 Distributed Evaluation: A Simple Query

Let us consider query *Q* that returns all *SLAPolicy* entries that have multiple *SLADSAction* descendants.

```
(d (dc=ISP, dc=com ? sub ? objectClass=SLAPolicy)
  ((count (dc=ISP, dc=com ? sub ? objectClass=SLADSAction)) ≥ 2))
```

and assume that the base entry with DN: *dc=ISP, dc=com* is managed by server *S1* (see Figure 1(b)). Current LDAP directory server implementations do not support such queries (distributed or not). Assuming that every LDAP server is extended to answer such queries locally (say using the centralized algorithms of [7]), and the topology tree of directory servers is known, we can evaluate *Q* in a distributed fashion as follows.

First notice that the query answer may consist of entries from each server: S1, S2, S3, S4; hence, the query can be evaluated by submitting some query to each server:

$$\text{Answer} = Q1@S1 \cup Q2@S2 \cup Q3@S3 \cup Q4@S4$$

Let us focus on query Q2 of this expression. It has to retrieve SLAPolicy entries having multiple SLADSAction descendant witness entries: a witness can be either local (in server S2) or remote (in server S3). Let v3 denote the number of SLADSAction entries in S3. Q2 needs to return two kinds of entries: (i) SLAPolicy entries that have multiple descendant witness entries in S2 itself, and (ii) SLAPolicy entries that have all v3 entries in S3 as descendants, and have at least 2 - v3 descendant witnesses in S2 itself. The first set of entries can be obtained by evaluating the original query Q locally at S2. For obtaining the second set of entries, we compute first the aggregate value expression (count (r3 ? sub ? objectClass=SLADSAction)) at server S3, where r3 is the partition root of S3. Then the following query is evaluated locally at S2:

```
(d (d (r2 ? sub ? objectClass=SLAPolicy) (exists (rr3 ? base ? objectClass=*))
  ((count (r2 ? sub ? objectClass=SLADSAction)) ≥ 2-v3))
```

where (d Q (exists Q')) = (d Q ((count Q') ≥ 1)), rr3 is the subordinate referral entry in server S2 that points to server S3, and v3 is the above aggregate value expression. (Note the use of base as a scope and of objectClass=* as a filter.) In summary, the final expression of the query Q2 can be determined only after some aggregate value expression is computed at server S3.

We will give a general procedure for complex queries in the next section.

4 Distributed Query Plan

We are given a hierarchical aggregate selection query Q , and a topology T consisting of the k servers S_1, \dots, S_k and their parent-child relationships. We show in this section how the client can generate a distributed plan for evaluating Q .

4.1 Distributed Plan Definition

Based on our discussion in the previous section, Q 's plan P_Q can be expressed as a union:

$$P_Q = Q_{S_1}@S_1 \cup \dots \cup Q_{S_k}@S_k \quad (2)$$

where each Q_{S_i} is a query returning the entries that satisfy Q and are managed by server $S_i, i = 1, \dots, k$. We call Equation (2) the *distributed query evaluation plan* for query Q , and call the queries Q_{S_1}, \dots, Q_{S_k} *server queries*. If S_i does not contribute entries to the answer, Q_{S_i} is empty. A single query to each server suffices, as long as the servers can evaluate hierarchical aggregate selection queries, as we constructively show later in this section.

Note that Equation (2) does not hold for general SQL queries on distributed relational databases. In that case, the different fields of an answer record may come from different servers (because of joins). This is a critical distinction, enabling us to derive much more efficient distributed directory query evaluation plans than in distributed relational databases.

4.2 Query Macros

Each server query $Q_{S_i}, i = 1, \dots, k$, in Equation (2) is meant to be evaluated locally, at server S_i . However, in general, the query expression may depend on other servers too. Namely each Q_{S_i} may have aggregate value expressions that need to be computed at one or more servers.

We call such an aggregate value expression a *macro* and write macros in square brackets, [...]. That is, macros have the form $[agg (Q'_{S_{j_1}}@S_{j_1} \cup \dots \cup Q'_{S_{j_m}}@S_{j_m}) valueExpr']$, where *agg* is one of `sum`, `min`, `max`. This is further decomposed into:

$$[(agg Q'_{S_{j_1}} valueExpr')@S_{j_1} op \dots op (agg Q'_{S_{j_m}} valueExpr')@S_{j_m}]$$

where *op* is one of `+`, `min`, or `max`, respectively. The last expression identifies the aggregate value expressions $[(agg Q'_{S_{j_i}} valueExpr')@S_{j_i}]$ to be computed at individual servers, the result of each of which is a single number.

Example 4.1 Consider again the hierarchical aggregate selection query Q of Section 3.3, abstracted and generalized as follows: $(d Q' ((count Q'') \geq v))$, where Q' and Q'' are simple LDAP queries that have the same base entry DN, managed by server S_1 , and the topology in Figure 1(b). Recall that server query Q_2 for S_2 depends on v_3 , the result of the aggregate value expression $(count Q'')$ at server S_3 . In our notation, v_3 is the result of a macro, and hence Q_2 becomes:

```
(| (d Q' ((count Q'') ≥ v))
  (d (d Q' (exists (rr3 ? base ? objectClass=*))
    ((count Q'') ≥ v - [(count Q'')@S3])))
```

where “|” is the boolean or operator (with a union semantics) ■

4.3 Distributed Query Plan Generation

We now describe how to generate a distributed query plan P_Q for an arbitrary hierarchical aggregate selection query Q and topology T . The plan is generated inductively on Q . We illustrate the most relevant cases for Q .

Atomic Selection Queries: Here query $Q = (\text{base-entry-DN} ? \text{scope} ? \text{atomic-filter})$.

Using the topology T , the client identifies the unique server S_i that manages `base-entry-DN`.

The plan generated depends on the scope. When the scope is `base`, the plan is $Q@S_i$;

when the scope is `sub`, the plan is $\cup(Q_{S_j}@S_j)$, where S_j iterates over S_i and all servers

that are descendants of S_i in the topology, and Q_{S_j} is as follows: $Q_{S_i} = Q$, and $Q_{S_j} =$

$(r_j ? \text{sub} ? \text{atomic-filter})$ for $j \neq i$ (r_j is the partition root of server S_j). We omit the case

when the scope is `one`, except to note that, in this case, both the base entry DNs and the

scopes of the queries sent to (some of the) servers that are children of S_i may be modified.

Boolean Selection Queries: We only consider $Q = (\& Q' Q'')$, where “&” is the boolean

and operator (with an intersection semantics). We generate inductively plans for Q' and Q'' :

$P_{Q'} = \cup_i(Q'_{S'_i}@S'_i)$, $P_{Q''} = \cup_j(Q''_{S''_j}@S''_j)$. Simple algebraic manipulations gives us the plan:

$$P_Q = \cup_{S'_i=S''_j}(\& Q'_{S'_i} Q''_{S''_j})@S'_i$$

Note that each server occurs at most once, which is a required property in Equation (2)

(see Section 4.1). The topology T was not needed here (but was needed inductively, in the

generation of $P_{Q'}$ and $P_{Q''}$). The plans for other boolean queries are similarly obtained.

Hierarchical Aggregate Selection Queries: There are four hierarchical aggregate se-

lection queries. We illustrate here only descendant queries, $Q = (d Q' \text{aggCond})$, where

the aggregate operator in aggCond (defined in Figures 2 and 3) is `sum`, and the relational

operator is \geq . To begin, we generate $P_{Q'}$ inductively, $P_{Q'} = \cup_{i=1}^k(Q'_{S'_i}@S'_i)$. Hence:

$$\begin{aligned} P_Q &= (d (\cup_{i=1}^k Q'_{S'_i}@S'_i) \text{aggCond}) = \cup_{i=1}^k(d (Q'_{S'_i}@S'_i) \text{aggCond}) \\ &= \cup_{i=1}^k(d Q'_{S'_i} ((\text{sum } Q'' \text{ valueExpr}'') \geq \text{valueExpr}')@S'_i) = \cup_{i=1}^k E_i \end{aligned}$$

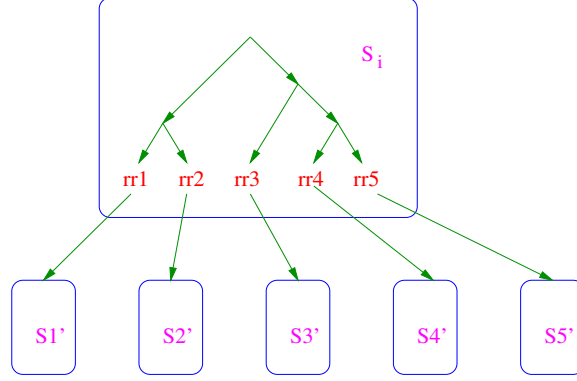


Figure 4: \mathcal{S} ranges over $2 \times 5 = 10$ out of the 2^5 subsets of $\{S1', \dots, S5'\}$.

This expression identifies the contribution of each server S_i , but it is not a plan yet, because *aggCond* may involve non-local computations. We consider next each E_i in turn. The meaning is that E_i returns entries in S_i satisfying Q'_{S_i} , whose set of descendants satisfying Q'' have an aggregate sum $\geq valueExpr'$. Here Q'' is still a distributed, non-local query, for which we now inductively construct the query plan $P_{Q''} = \bigcup_{j=1}^k (Q''_{S_j} @ S_j)$. Since we consider the “descendant” operator, d , we can replace all Q''_{S_j} for which S_j is neither S_i nor a descendant of S_i with an empty query. We have three cases to consider.

Case 1: $P_{Q''}$ at S_i : In the simplest case, $P_{Q''}$ involves only a computation at S_i (that is, only Q''_{S_i} is nonempty). Then we have:

$$E_i = (d Q'_{S_i} ((\text{sum } Q''_{S_j} \text{ valueExpr}'') \geq \text{valueExpr}')) @ S_i$$

which is a local computation at S_i .

Case 2: $P_{Q''}$ at Children Servers: In this case, $P_{Q''}$ involves computations at servers that are children of S_i . So assume $P_{Q''}$ involves computations at S'_1, \dots, S'_m , all of which are S_i 's children; this is depicted in Figure 4, with $m = 5$. We denote $v_j = (\text{sum } Q''_{S'_j} \text{ valueExpr}'') @ S'_j$,

an aggregate value expression computed locally at $S'_j, j = 1, \dots, m$. For every subset of children $\mathcal{S} = \{S'_{i_1}, \dots, S'_{i_n}\}$, we need to consider the contribution to E_i of those entries in S_i having all entries from servers in this set as their descendants. This contribution is:

$$F_i(\mathcal{S}) = (d (d Q'_{S_i} \mathcal{S})((\text{sum } Q''_{S_i} \text{ valueExpr''}) + v_{i_1} + \dots + v_{i_n} \geq \text{valueExpr'})), \text{ where}$$

- $(d Q'_{S_i} \mathcal{S})$ abbreviates the expression² $(\& (d Q'_{S_i} (\text{exists } rr_{i_1})) \dots (d Q'_{S_i} (\text{exists } rr_{i_n})))$, which selects entries in S_i satisfying Q'_{S_i} and having all servers in \mathcal{S} as descendants.
- The values v_{i_1}, \dots, v_{i_n} become macros in $F_i(\mathcal{S})$, as before.

The entire expression E_i becomes: $E_i = (\bigcup_{\mathcal{S}} F_i(\mathcal{S}))@S_i$. We now substitute for E_i , and obtain the distributed plan for Q . Once macros for v_1, \dots, v_m are substituted with numbers, E_i becomes a local expression that can be evaluated entirely at S_i . Note that this plan relies on directory partitions having unique partition roots.

Finally, we need to consider the range of subsets \mathcal{S} : only $2m$ out of the 2^m subsets need to be considered. For an intuitive illustration, we refer to Figure 4, showing an example with $m = 5$. Given the m DNs, rr_1, \dots, rr_m , the client first constructs a hierarchy, as suggested in Figure 4, using common suffixes of the m DNs, and its result is a tree whose leaves are rr_1, \dots, rr_m . Then, we only need to consider subsets \mathcal{S} that are the set of leaves of some subtree, plus $\mathcal{S} = \emptyset$. There are $2m - 1$ subtrees, hence \mathcal{S} ranges over $2m$ sets.

Case 3: $P_{Q''}$ at Descendant Servers: Finally, we turn to the case when $P_{Q''}$ involves computations at servers that are indirect descendants of S_i . Here, we still consider S_i 's children S'_1, \dots, S'_m and we proceed as above, with the only difference in the way we compute

²We use rr_{i_j} as an shorthand for the selection query $rr_{i_j} ? \text{base} ? \text{objectClass} = *$.

v_j . Instead of $(\text{sum } Q''_{S'_j} \text{ valueExpr''})@S'_j$, now we have to also add the aggregate values of all of S'_j 's descendants, S''_1, S''_2 , etc. The value v_j is given by:

$$(\text{sum } Q''_{S'_j} \text{ valueExpr''})@S'_j + (\text{sum } Q''_{S''_1} \text{ valueExpr''})@S''_1 + (\text{sum } Q''_{S''_2} \text{ valueExpr''})@S''_2 + \dots$$

Example 4.2 Consider again the query $Q = (d \ Q' \ ((\text{count } Q'') \geq v))$ in Example 4.1, for the topology in Figure 1(b), whose plan has the form $Q1@S1 \cup Q2@S2 \cup Q3@S3 \cup Q4@S4$.

Our algorithms generate expressions for $Q1, Q2, Q3, Q4$, resulting in the following plan:

$$\begin{aligned} &(| \ (d \ Q' \ ((\text{count } Q'') \geq v)) \\ & \ (d \ (d \ Q' \ \{S2, S4\})((\text{count } Q'') + [(\text{count } Q'')@S2] + [(\text{count } Q'')@S3] + \\ & \ \ [(\text{count } Q'')@S4] \geq v)) \\ & \ (d \ (d \ Q' \ \{S2\}) \ ((\text{count } Q'') + [(\text{count } Q'')@S2] + [(\text{count } Q'')@S3] \geq v)) \\ & \ (d \ (d \ Q' \ \{S4\}) \ ((\text{count } Q'') + [(\text{count } Q'')@S4] \geq v)))@S1 \\ & \ \cup \\ & (| \ (d \ Q' \ ((\text{count } Q'') \geq v)) \\ & \ (d \ (d \ Q' \ \{S3\}) \ ((\text{count } Q'') + [(\text{count } Q'')@S3] \geq v)))@S2 \\ & \ \cup \\ & (d \ Q' \ ((\text{count } Q'') \geq v))@S3 \\ & \ \cup \\ & (d \ Q' \ ((\text{count } Q'') \geq v))@S4 \end{aligned}$$

Despite the lengthy expression, the plan has a simple structure. It contains three macros, $[\text{count } Q'']@S2$, $[\text{count } Q'']@S3$, and $[\text{count } Q'']@S4$ that need to be evaluated first, at servers $S2, S3$, and $S4$ respectively. The resulting numbers are then plugged into the query expressions, and the four queries $Q1, Q2, Q3, Q4$ are then submitted independently to the four servers. ■

4.4 Aggregate Value Cache

In a query plan, the same macro may occur multiple times. In Example 4.2, $(\text{count } Q'')@S3$ occurs both in $Q1$ and in $Q2$. Moreover, in $Q1$ it occurs twice. A naive scheduling would submit that expression multiple times to $S3$. We use an *aggregate value cache* at the client to store the values returned by aggregate value expressions, thus avoiding repeated executions. As we shall see in our experiments, this results in significant performance improvements.

It is important to note here that the benefit of an aggregate value cache can be realized with a *single* hierarchical aggregate selection query. If the distributed directory has n partitions and the hierarchical aggregate selection query has nesting depth k , then an aggregate value cache of size $O(n * k)$ suffices to cache the results of all aggregate value expressions in the distributed evaluation plan. For reasons of consistency, we assume that the result of a cached expression does not change during the evaluation of a query. Note that since we do not use the aggregate value cache in a traditional way (across several queries), we do not have to deal with issues of cache management, such as admission and replacement policies.

4.5 Discussion

Our query plan construction technique can be extended without much difficulty to other hierarchical aggregate selection queries. We do not discuss them in detail for reasons of space. We have the following upper bounds on the total number of messages exchanged, and the total size of messages exchanged during the execution of any query plan P_Q .

Theorem 4.1 *Consider a hierarchical aggregate selection query Q with nesting depth k , and query plan P_Q evaluated against a distributed directory with n partitions (servers). Then,*

the total number of rounds of messages exchanged between the directory client (that uses an aggregate value cache) and the directory servers is $O(n * k)$. The total amount of data exchanged in these messages is $O(|Ans(Q)| + n * k)$, where $Ans(Q)$ is the answer to Q . ■

Note that, as in the case of boolean LDAP query evaluation by referrals, the total communication cost of distributed evaluation of hierarchical aggregate selection queries is *independent* of the total size of the directory data. However, unlike the case of boolean LDAP query evaluation by referrals (see Proposition 3.1), the number and size of messages is a function of the nesting depth k of the query.

A final remark worth making is that our techniques for the distributed evaluation of hierarchical aggregate selection queries allow per-server computation costs to be optimized *independently* of communication costs, in a data-independent manner. This is in contrast to the distributed evaluation of SQL queries, which involves a substantial, data-dependent, tradeoff between communication and computation; the use of semijoin-style optimizations in distributed SQL query evaluation, for example, attempts to reduce communication cost at the expense of increased per-server computation cost [3, 9].

5 Experimental Results

The aim of the experiments presented here is to show the feasibility of our techniques for distributed directory query processing. Our results are based on a real implementation that modified the OpenLDAP directory server [2], and enhanced the LDAP client library.

5.1 Implementation Alternatives

We performed two sets of enhancements to OpenLDAP in order to handle hierarchical aggregate selection queries of the form ($d Q' (\text{exists } Q'')$). The first set of enhancements, needed for a single site evaluation, consists of (i) extending the LDAP protocol to understand the new query syntax, and (ii) extending the LDAP server with the appropriate algorithms and index structures to compute the hierarchical aggregate selections. The second set of enhancements, needed for the distributed evaluation of these queries, consists of building a client application that takes a user query and the topology tree information, selects the relevant portion of this topology and then generates the appropriate distributed query plan, as described in previous sections.

An alternative approach is to enhance *only* the LDAP client to deal with hierarchical aggregate selection queries. In this case, the LDAP server and the LDAP protocol need not be extended. Since the LDAP server understands only LDAP queries, the client would be responsible for combining the results (to a sequence of LDAP queries) to answer the user query. This approach is clearly much less efficient than our approach of enhancing the server, since it results in considerably more data traffic between the client and the server. We do not consider this approach further.

5.2 Data Sets and Metrics

We used four topology configurations for our experiments: (a) left-deep skinny, where each server has two children servers (and only one of which may (recursively) have children), varying the depth of the tree from 0 (1 server) to 10 (21 servers); (b) left-deep bushy, where

each server has five children servers (and only one of which may (recursively) have children), varying the depth of the tree from 0 (1 server) to 4 (21 servers); (c) balanced skinny, which is a complete balanced binary tree of servers, varying the depth from 0 (1 server) to 4 (31 servers); and (d) balanced bushy, which is a complete balanced 5-ary tree of servers, varying the depth from 0 (1 server) to 2 (31 servers).

Since the aim of our experiments is to better understand the feasibility of highly distributed directory query evaluation for different server topologies, and not the performance of a stand-alone directory server, each of our directory servers had few (100) entries.

The timing results of our experiments are normalized to highlight the performance trends of distributed directory query processing, instead of focusing on absolute numbers (which would be more relevant if we were focusing on the performance of stand-alone directory servers). Subfigures in the same figure use the same normalization factor to enable easy comparison, while different figures use different normalization factors.

5.3 Using an Aggregate Value Cache

We now demonstrate the utility of an aggregate value cache for achieving a scalable, distributed evaluation of hierarchical aggregate selection queries.

Queries: For most of the experiments demonstrating the utility of an aggregate value cache, we used the simple hierarchical aggregate selection query ($d Q' (\text{exists } Q'')$), where Q' and Q'' are boolean LDAP queries with the same base DN. The queries were varied to return different numbers of answers. For one experiment (which we explicitly identify), we used a hierarchical aggregate selection query, designed to have an empty result, with a

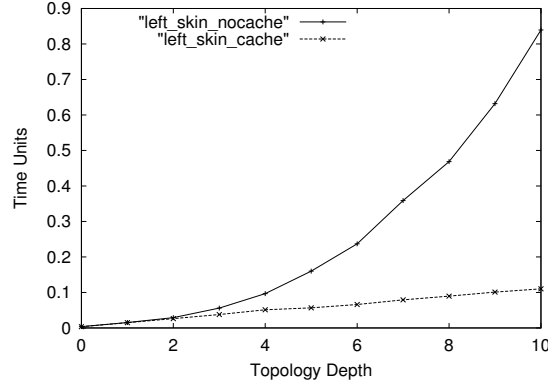
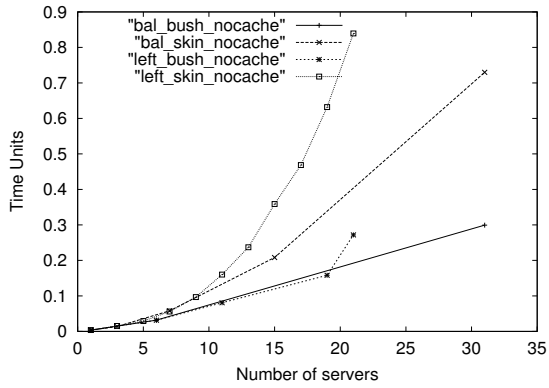


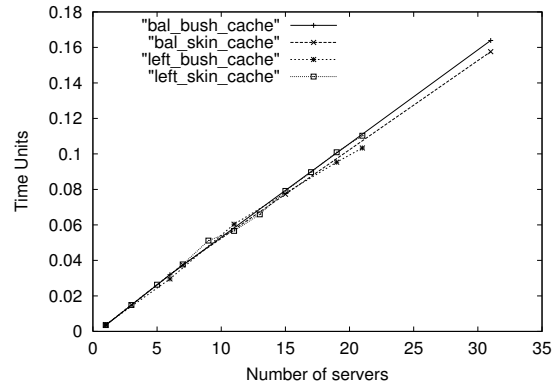
Figure 5: Utility of an Aggregate Value Cache

nesting depth of 2 ($d Q' (\text{exists } (d Q'' (\text{exists } Q'''))))$), where Q' , Q'' and Q''' are boolean LDAP queries with the same base DN.

Results: The results we obtained can be classified into three sets. The first set is given in Figure 5. Here, the simple hierarchical aggregate selection query was used. While both Q' and Q'' have non-empty results, the query as a whole has an empty result. As a consequence, our algorithms have to construct complex server queries for each non-leaf directory server in the distributed directory. This figure shows a simple graph containing two curves, representing the performance of our algorithm with and without the use of an aggregate value cache. The graph was generated for the left-deep skinny topology, varying the depth of the tree from 0 to 10 (the number of servers varies from 1 to 21). Without the use of the aggregate value cache, the same aggregate value expression can be computed multiple times at a given server. These two curves show the quadratic nature of the no cache approach and the linear nature of the approach with the aggregate value cache, as a function of the maximum path length between the server that manages the base entry DN of the query, and any other server that needs to be contacted.

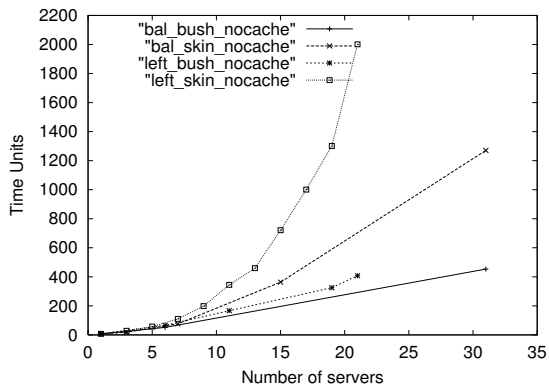


(a) No Aggregate Value Cache

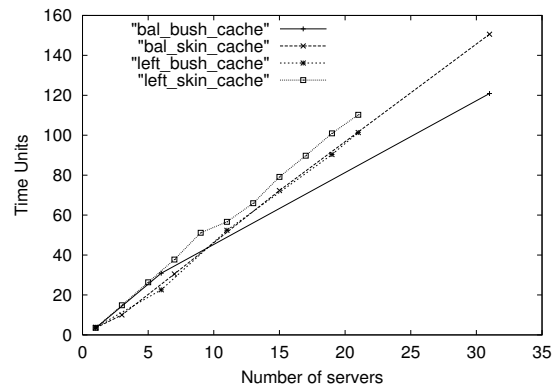


(b) With an Aggregate Value Cache

Figure 6: Scalability of Distributed Query Evaluation (0 answers)



(a) No Aggregate Value Cache



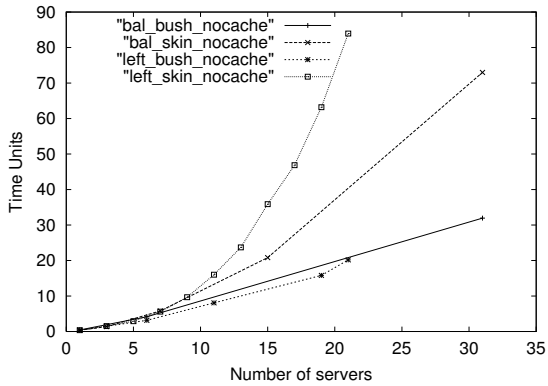
(b) With an Aggregate Value Cache

Figure 7: Scalability of Distributed Query Evaluation (100 answers)

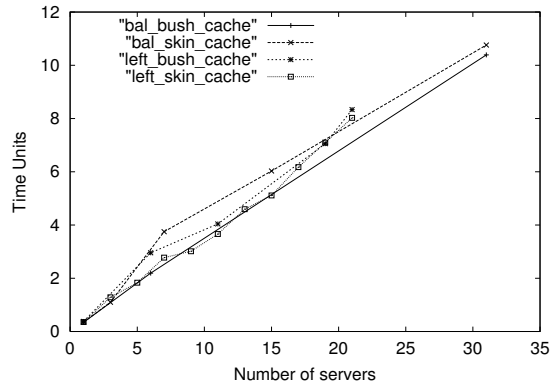
The second set of experiments shows the performance results for distributed directory query evaluation with and without the use of an aggregate value cache for the simple hierarchical aggregate selection query with a varying number of answers. In the first case (Figure 6), the query had no answer, in the second case (Figure 7), the query had 100 answers. We experimented with queries that returned intermediate numbers of answers as well; the results are similar and are omitted.

Figure 6(a) shows the performance results for the evaluation of the query without an aggregate value cache. In this case, the query cost is proportional to the total number of relevant servers in the topology and to the square of relevant topology's depth. Figure 6(b) shows the performance of the same query with the use of an aggregate value cache, demonstrating the insensitivity to the topology. Figure 7 shows similar results as Figure 6. The only difference is in the time overhead of selecting a higher number of query answers each time. However, the trends remain the same.

Finally, the third set of experiments in this category is presented in Figure 8. The aim of this experiment is to show that the utility of an aggregate value cache increases with the complexity of the hierarchical aggregate selection query. The query we used for this experiment is the query with a nesting depth of 2 (mentioned above). In order to focus on the distribution overheads of our evaluation, the query result was chosen to be empty. The interesting remark to be made here is that the trends that we observed with the simpler query remain the same. In addition, the use of an aggregate value cache in this last experiment is even more interesting since the time difference between the curves in Figure 8(a) and those in Figure 8(b) is more significant than that for the simpler query (Figure 6).



(a) No Aggregate Value Cache



(b) With an Aggregate Value Cache

Figure 8: Scalability of Distributed Query Evaluation with a Complex Query

In all cases, the following two observations hold. First, distributed evaluation of directory queries, using an aggregate value cache, is *robust*, in that the cost is independent of the specific topology, but depends only on the number of relevant servers. Second, the evaluation strategy is *scalable* with respect to the number of servers, as is evident from the linear nature of the curves with an aggregate value cache.

5.4 Separating Topology Identification

The LDAP strategy of distributed evaluation by referrals discovers the topology tree dynamically, while evaluating the query at each server. Our approach relies on identifying the entire topology tree first, independent of the location of the query base, and then generating and executing the query at each relevant server in the topology tree; this is the case for both LDAP queries and hierarchical aggregate selection queries. To amortize the cost of topology tree identification, a “topology” query is asked at the beginning of each session (i.e., when an

LDAP client is created), and the fetched topology tree is used for all queries in that session (i.e., until the LDAP client is killed). All session state, including the topology tree and the per-query aggregate-value cache, is stored at the LDAP client.

In this section, we run two sets of experiments to show the utility of separating the identification of the topology tree from the query plan generation and execution, using simple LDAP queries. The first set of experiments shows the benefit of identifying the topology for an LDAP query. The second set of experiments quantifies the advantages of amortizing the cost of topology identification across multiple queries in the same session.

In the sequel, we call the case where the topology of directory servers is discovered while a query is evaluated (by referrals) as the *notree* approach. We call the case where the topology tree is identified in the beginning of a user session as the *tree* approach.

5.4.1 Benefit of Topology Identification for a Boolean LDAP Query

In this experiment, we compare the *notree* and *tree* approaches for a boolean LDAP query. When using the *notree* approach, there is a first phase during which the server that manages the base entry DN of the submitted LDAP query has to be located. The optimal situation for the *notree* approach is to hit the server that contains the query base entry DN in the beginning. To ensure a fair comparison of our evaluation strategy with the *notree* approach, we make sure that the LDAP query is submitted to the server that manages its base.

Queries: Since we do not want any query answer traffic to mask the differences between the LDAP approach and ours, we used LDAP queries that have no matching answers; however, the evaluation techniques have to search the relevant sub-topology to determine this outcome.

Thus, the time taken for communicating the results from the server to the client can be attributed only to the distribution overheads.

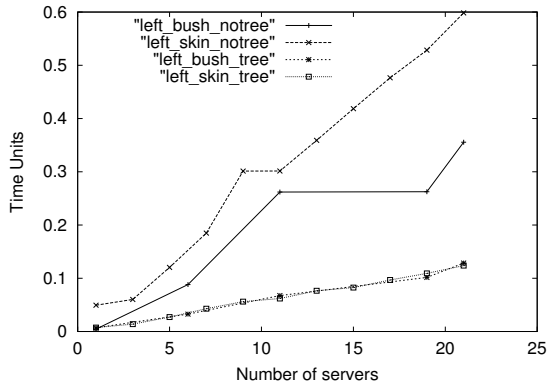
Results: Figure 9 shows the experimental results on our topology configurations comparing the evaluation that builds and uses the topology tree to the evaluation mechanism using referrals (notree). The graph on the left shows the numbers for the two left-deep topologies, the graph on the right shows the numbers for the two balanced topologies; both are normalized using the same factor. On the x axis, we plot the number of servers, on the y axis, we measure the (normalized) times for contacting all the servers.

In both cases, the two curves representing the notree approach are above the ones representing the tree approach. The benefits only increase with a larger number of servers; this is emphasized by the fact that in both graphs, the absolute value of the differences between the notree and the tree curves grows linearly with the number of servers.

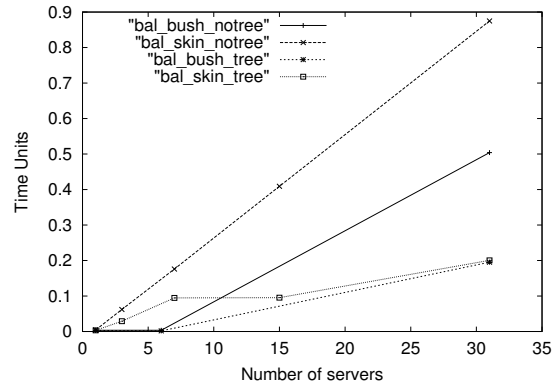
Further, the graphs show that the notree approach is highly sensitive to the distribution of the servers. As we increase the number of servers, the evaluation times for skinny and bushy approaches begin to diverge, demonstrating the sensitivity to the depth of the topology. However, using the topology tree makes the evaluation cost of LDAP queries insensitive to the topology; as the number of servers increases, the evaluation times for the skinny and bushy approaches begin to *converge*. This is a measure of the robustness of our technique.

5.4.2 Amortizing Topology Identification Time

In order to study the benefit of amortizing the topology identification time, we run two sets of experiments. Our results are similar across different topologies. Therefore, we chose to show



(a) Left-deep Topologies



(b) Balanced Topologies

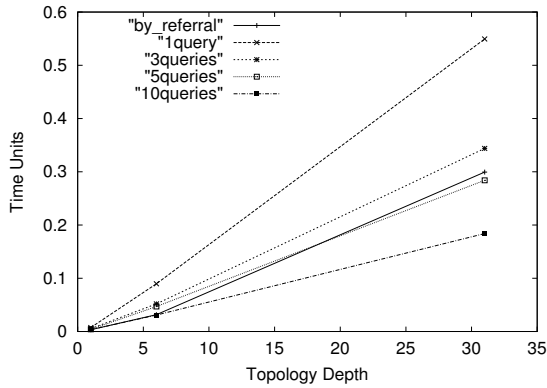
Figure 9: Using a Topology Tree

results obtained for balanced bushy topologies only. Figure 10(a) contains the results for the first set of experiments. Figure 10(b) shows the results of the second set of experiments.

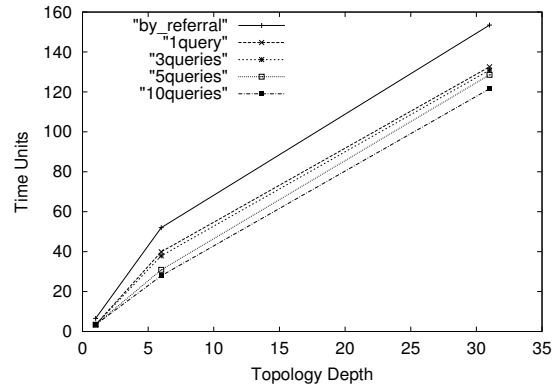
Queries: As in the previous experiments, we ran boolean LDAP queries, but with varying numbers of answers (0 and 100). Both Figures 10(a) and 10(b) contain 5 curves. We report the average time of running one query per user session using the notree approach. Then, we report the average time of running one query per session using the tree approach, then three queries, then five queries and finally ten queries per session using the same approach.

Results: In the tree approach, the topology identification time is amortized across the set of all queries executed in the same session. We want to understand how “soon” does this amortization pay off in a session.

The graph given in Figure 10(a) shows that when five queries are executed in the same session, the cost of identifying the topology prior to query execution is amortized. This



(a) Queries with an Empty Answer



(b) Queries with 100 Answers

Figure 10: Amortizing Topology Identification Time

means that if a user runs at least five queries in the session, the benefit of identifying the topology becomes clearly visible. Recall that Figure 10(a) reports evaluation times for queries returning an empty result. In the case of queries returning non-empty results (case of Figure 10(b) where queries return 100 answers), the “amortization” happens much sooner. In fact, this case can be considered as “an extreme” case since all the curves obtained with the tree approach are “below” the one obtained with the “notree” approach. In fact, if the user runs a single query in the session, the cost of identifying the topology in the beginning of this session is amortized. This is mainly due to the parallelization achieved during the evaluation of the server queries.

In conclusion, we can say that we expect the cost amortization to happen early in a user session and thus, recommend performing topology identification prior to query evaluation. Moreover, if this topology is known to be static, it could be used across multiple user sessions to save even more time.

6 Conclusion

Directories that are used in the network infrastructure, like in the DEN initiative, would greatly benefit from an efficient, distributed evaluation of hierarchical aggregate selection queries. In this paper, we demonstrate the practicality of such a distributed evaluation.

We showed, analytically and experimentally, that by (a) using an a-priori knowledge of the topology to quickly identify directory servers that are relevant to a query, and (b) maintaining and taking advantage of a small aggregate value cache at the directory client, both LDAP queries and the richer hierarchical aggregate selection queries of [7] can be evaluated efficiently in a highly distributed fashion. In particular, we showed that our mechanism of distributed evaluation is both (i) scalable, and (ii) robust, in that the evaluation cost grows linearly with the number of query relevant directory servers, and that this cost is independent of the specific topology of servers.

In addition to being distributed, directory data is usually highly replicated for reasons of availability and performance. An important direction of future work is to take advantage of replicated directory data, in particular knowledge of the replication topology, for improving the efficiency of distributed evaluation of hierarchical aggregate selection queries.

Acknowledgements

We would like to thank Juliana Freire, and the anonymous referees for their many helpful comments on various drafts of the paper.

References

- [1] Directory enabled networks ad hoc working group. <http://www.murchiso.com/den/>.
- [2] Open LDAP: Community developed LDAP software. Available from <http://www.openldap.org/>.
- [3] P. A. Bernstein and N. Goodman. Power of natural semijoins. *SIAM Journal of Computing*, 10(4):751–771, November 1981.
- [4] R. Chaudhury, E. Ellesson, S. Kamat, J.-C. Martin, G. Powers, R. Rajan, D. Verma, and R. Yavatkar. Directory schema for service level administration of differentiated services and integrated services in networks. Draft submitted to Directory Enabled Networks Ad Hoc Working Group. Available from <http://www.murchiso.com/den/specifications/den-draft-ellesson-sla-schema-02.txt>, 1998.
- [5] T. Howes and M. Smith. *LDAP: Programming directory-enabled applications with lightweight directory access protocol*. Macmillan Technical Publishing, Indianapolis, Indiana, 1997.
- [6] T. Howes, M. Smith, and G. S. Good. *Understanding and deploying LDAP Directory Services*. Macmillan Technical Publishing, Indianapolis, Indiana, 1999.
- [7] H. V. Jagadish, L. V. S. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 133–144, Philadelphia, PA, June 1999.
- [8] D. Kossman. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
- [9] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558–560, 1990.
- [10] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.
- [11] J. Strassner. *Directory enabled networks*. Macmillan Technical Publishing, Indianapolis, Indiana, 1999.
- [12] M. Wahl, T. Howes, and S. Kille. Lightweight directory access protocol (v3). Request for Comments 2251. Available from <ftp://ftp.isi.edu/in-notes/rfc2251.txt>, Dec. 1997.