Paper Number P027

Querying Network Directories

H. V. Jagadish
AT&T Labs–Research
jag@research.att.com

Laks V. S. Lakshmanan
Concordia University
laks@cs.concordia.ca

Tova Milo
Tel-Aviv University
milo@math.tau.ac.il

Divesh Srivastava*
AT&T Labs–Research
divesh@research.att.com

Dimitra Vista†
AT&T Labs–Research
vista@research.att.com

Abstract

Hierarchically structured directories have recently proliferated with the growth of the Internet, and are being used to store not only address books and contact information for people, but also personal profiles, network resource information, and network and service policies. These systems provide a means for managing scale and heterogeneity, while allowing for conceptual unity and autonomy across multiple directory servers in the network, in a way far superior to what conventional relational or object-oriented databases offer. Yet, in deployed systems today, much of the data is modeled in an ad hoc manner, and many of the more sophisticated "queries" require navigational access.

In this paper, we develop the core of a formal data model for network directories, and propose a sequence of efficiently computable query languages with increasing expressive power. The directory data model can naturally represent rich forms of heterogeneity exhibited in the real world. Answers to queries expressible in our query languages can exhibit the same kinds of heterogeneity. We present external memory algorithms for the evaluation of queries posed in our directory query languages, and prove the efficiency of each algorithm in terms of its I/O complexity. Our data model and query languages share the flexibility and utility of the recent proposals for semi-structured data models, while at the same time effectively addressing the specific needs of network directory applications, which we demonstrate by means of two representative real-life examples.

*Contact author. AT&T Labs–Research, Room A-105, 180 Park Avenue, Florham Park, NJ 07932, USA, Tel: +1-(973)-360-8776, Fax: +1-(973)-360-8871, E-mail: divesh@research.att.com.

†Work done while at AT&T Labs–Research.

1 Introduction

Hierarchically structured directories have recently proliferated with the growth of the Internet, and a large number of commercial directory server implementations are now available (see [20] for a survey). They are currently being used to store address books and contact information for people, enabling the deployment of a wide variety of network applications such as corporate white pages and electronic messaging. The Internet Engineering Task Force (IETF) has recently standardized the popular Lightweight Directory Access Protocol (LDAPv3) for modeling and querying network directory information, as well as accessing network directory services [30, 29, 31, 17, 19]. An LDAP-based network directory can be viewed as a highly distributed database, in which the directory entries are organized into a hierarchical namespace and can be accessed using database-style search functions.

More recently, LDAP is being proposed as the basis of the *directory enabled networks* (DEN) initiative for representing profiles of network users, devices, applications and services, as well as policies for the overall management of the network, in a directory (see, e.g., [12, 15, 26]). We demonstrate, using two significant applications in Section 2, that DEN applications use directories in ways that are considerably more complex than the current generation of directory enabled applications.

Our thesis in this paper is that, although it is largely appropriate for the current generation of management and browser applications that provide read/write interactive access to LDAP directories, *the LDAP query language is woefully inadequate for the new generation of DEN applications.* With LDAP, DEN applications would have to specify not only *which* directory entries need to be accessed, but also *how* to access them, using long sequences of queries. Three decades of research in high-level database query languages has proved the advantage of *declarative* languages, demonstrating that applications should merely have to specify which directory entries need to be accessed, leaving the task of determining how to efficiently access directory entries to the query evaluation engine of the directory.

In this paper, we seek to bridge the considerable gap between the directory query requirements of DEN applications and the constructs provided by the LDAP query language, and make the following contributions:

- We present a formal description of the core of a scalable network directory data model, in Section 3, in the spirit of LDAP and DNS. We illustrate that the directory data model can naturally represent the rich forms of heterogeneity needed by network directory applications.
- We devise a sequence of efficiently computable query languages, in Sections 4–7, retaining the core LDAP philosophy of incurring low resource requirements. Each language in this sequence illustrates a specific class of significant queries for DEN applications not supported by the current LDAP standard. Answers to queries can exhibit the same kinds of heterogeneity as directory instances.
- We compare the expressive power and computational complexity of the query languages and evaluation algorithms we propose, as well as the current LDAP standard, in Section 8. Our central results are that: (a) the query languages exhibit a strict hierarchy of expressive power; (b) queries written in all but the most expressive language can be evaluated with time and I/O complexities that are linear in the size of the inputs to the query; and (c) queries written in the most expressive language can be evaluated with time and I/O complexities that are proportional to $N * \log N$, for inputs of size N.

After a survey of related work in Section 9, we conclude with a discussion in Section 10.

2 Motivation: Directory Enabled Networks

A directory enabled network (DEN) represents profiles of network users, applications and services, as well as policies for the overall management of the network, in a directory (see, e.g., [12, 15, 26]). In this section, we introduce and motivate two running examples, to be used throughout the paper, both from actual DEN applications that we have studied.

Example 2.1 [Supporting Quality of Service (QoS) in the Core Network]
Internet Service Providers (ISPs) and corporate administrators use Service Level Agreements (SLAs) to define contracts between the user and the network. With the introduction of such agreements, the network now needs to discriminate between different packets to provide multiple service levels. This is achieved using policies that could specify, for example, the service category to be employed for a particular application, how much bandwidth is allocated to a particular flow category, the maximum number of flows to be supported between two subnets, etc. A *policy* defines the desired behavior between multiple objects, and has two principal components: a profile and an action. The profile identifies the objects that are relevant to the policy, and the action specifies the desired behavior; both are defined by the values of a collection of attributes. For concreteness, we focus on the proposal by Chaudhury et al. [11] for representing such policies in a directory.

Directory Contents: The directory is a repository of *policies*, grouped by administrative domains, that specify constraints for access to network resources based on administrative criteria. A simple example is "Drop all packets from the source address 204.178.16.5". The policy's profile specifies the sub-stream of packets that the policy applies to, and the policy's action specifies the treatment of packets so identified. In the above example "source 204.178.16.5" constitutes the profile, while the action enforced is "Drop".

Since the network administrative policy in force is constituted by the totality of these policies, it is important to ensure that these policies, considered together, define an unambiguous treatment of packets. A policy conflict occurs when two or more policies in the directory specify conflicting actions for the same packet; such conflicts must be resolved before populating the directory. [11] describes two mechanisms to indicate which policy takes precedence in case of conflict. First, a *priority attribute* is associated with each policy, and in case of conflicts the highest priority policy is applied; priority attributes allow an unambiguous specification of policies using a total order. Second, an *exception attribute* resolves the conflict between two policies in the region of their overlap; if Policy 2 is an exception for Policy 1, then Policy 2 is applied to all packets matching the profiles of both policies; exception attributes allow for easy insertion and deletion of policies in the directory.

Figure 12, in the appendix, shows a fragment of a sample directory of such policies, but perusal of this figure is best deferred until the reader has a better perspective on the data model (presented in Section 3).

Directory Queries and Answers: Queries are issued by policy enforcement entities, for example, hosts, routers, firewalls, and proxy servers, that encounter (data or signaling) packets flowing across the network.

When packets of a certain type are encountered, e.g., RSVP message, or IP packet bearing a TCP connect request, the enforcement entity provides: (a) the attributes of the packet (source address and port, destination address and port, protocol number, etc.), and (b) the current time, as the profile to be matched against the directory. It asks for the actions that are specified by the policies that match the given profile, such that: (a) no higher priority policies apply to the packet, and (b) the policies have no exceptions of the same priority that apply to the packet.

These actions are then applied by the enforcement entities in conditioning the packet stream. ∎

Example 2.2 [Supporting Location and Device Independent Access]
In order to reach the vast majority of telephone subscribers, a caller needs to know the network address (telephone number) of the terminal closest to the subscriber's current location, for example, his office phone, home phone, car phone, etc. The Telephony Over Packet networkS (TOPS) project [2, 3] has the goal of providing a simple dial-by-name capability that allows subscribers to move between terminals or to use mobile terminals while being reachable by the same name. As our second DEN application, we briefly describe the TOPS directory requirements.

Directory Contents: Each TOPS subscriber is represented in the network directory by a directory entry that contains the subscriber *profile* (e.g., full name, address, authentication credentials, etc.), and a set of prioritized subscriber *policies* that determine how the subscriber can be reached.

Each policy consists of a *query handling profile* (QHP), that allows subscribers to control access by specifying who can reach them, and a set of *call appearances*, representing the different ways in which the subscriber can be reached by the caller who satisfies the QHP. A call appearance is typically associated with a terminal device or server and consists of a set of attributes that identify the type, network address and terminal capabilities of the call appearance.

Subscriber profiles are created at the time of TOPS service subscription, while subscriber policies can be created and modified dynamically.

Figure 11, in the appendix, shows a fragment of a sample directory of such query handling profiles and call appearances. Again, perusal of this figure is best deferred until the reader has a better perspective on the data model (presented in Section 3).

Directory Queries and Answers: In order to call a subscriber, the calling application queries the directory using the logical name of the subscriber to obtain his call appearances. In addition, callers may also supply their own logical name, the types of media to be included in the call, the capabilities of the calling terminal, etc.

The caller provided information, along with the time of day, the compatibility between the caller's and callee's terminal capabilities, etc., are matched against the query handling profiles of the subscriber's policies. The response to such a query is the set of call appearances where the subscriber can be reached, corresponding to the highest priority policy that matches the given information. This provides subscribers with customizability, and considerable control over the privacy of their information.

When the calling TOPS application receives this information, it may use the call appearances directly, taking into consideration user/application policy, or it may present the call appearances to the caller, who can choose from amongst the call appearances based on his current needs. ∎

3 The Network Directory Data Model

In this section, we present a formal description of the core of a scalable network directory data model, based on LDAP [30] and DNS [24], that is particularly suitable for DEN applications. We then revisit our motivating applications, and illustrate the modeling of their data using the network directory data model.

3.1 Directory Schema

We assume pairwise disjoint infinite sets $\mathcal{A}, \mathcal{C}$ of attributes and class names, as well as a set $\mathcal{T}$ of type names. Each type $t \in \mathcal{T}$ has an associated domain, denoted $dom(t)$. We use $dom(\mathcal{T})$ to abbreviate $\bigcup_{t \in \mathcal{T}} dom(t)$. Without loss of generality, we assume that the attribute `objectClass` is in $\mathcal{A}$ and that $\mathcal{T}$ includes the basic types `string` and `int`.[1] We also assume the existence of a complex type `distinguishedName` $\in \mathcal{T}$, whose domain consists of sequences of sets of pairs in $\mathcal{A} \times dom(\mathcal{T})$. As explained later, the values of this special type are used as keys to identify directory entries.

Definition 3.1 (Directory Schema) A *directory schema* is a 4-tuple $S = (C, A, \tau, \alpha)$ where: (a) $C \subset \mathcal{C}$ is a finite set of class names; (b) $A \subset \mathcal{A}$ is a finite set of attributes, such that `objectClass` $\in A$; (c) $\tau : A \rightarrow \mathcal{T}$ is a function that associates a type with each attribute, such that $\tau($`objectClass`$) =$ `string`; and (d) $\alpha : C \rightarrow 2^A$ is a function that associates a set of attributes with each class name.

For a class $c \in C$, we call $\alpha(c)$ the set of *allowed attributes* of c. ∎

As a schema element, the notion of a class plays a role similar to that of a relation in the relational model, or a class in the object-oriented model. A key difference stems from the decoupling of attributes from classes: since the type of an attribute is defined independently of the classes having the attribute, occurrences of the same attribute in multiple classes all share the same type.

3.2 Directory Instance

Just as the relational model uses relations as a single uniform data structure to represent information, our model uses a *forest* as a single data structure. We call nodes of this forest *directory entries.* Intuitively, each entry has a *distinguished name* and may "hold" information in the form of a set of (attribute, value) pairs. These intuitions are formalized below. We assume an infinite set $\mathcal{R}$ of objects called directory entries.

Definition 3.2 (Directory Instance) A *directory instance* of a directory schema $S = (C, A, \tau, \alpha)$ is a 4-tuple $I = (R, class, val, dn)$, such that: (a) $R \subset \mathcal{R}$ is a finite set of directory entries; (b) the function $class : R \rightarrow 2^C$ associates with each directory entry a non-empty set of classes from C, to which it belongs; (c) $val : R \rightarrow 2^{A \times dom(\mathcal{T})}$ is a function that associates with each directory entry a set of (attribute, value) pairs[2], s.t. the following conditions are satisfied:

[1] Commercial directory servers, such as Netscape Directory Server 3.1, additionally provide types to deal with telephone numbers, binary data, and distinguish between case-sensitive and case-insensitive strings.

[2] Note that several pairs with the same attribute name may belong to the set, hence an attribute may have multiple values in a directory entry.

1. For each entry $r \in R$, $val(r)$ contains a pair (a, v) iff there exists a class name $c \in class(r)$ s.t. $a \in \alpha(c)$, $\tau(a) = t$ and $v \in dom(t)$. That is, $val(r)$ contains this pair iff the attribute a is an allowed attribute for at least one of r's classes, and the value v is of the right type.

2. For every class name $c \in C$, for every directory entry $r \in R$, $(\texttt{objectClass}, c) \in val(r)$ iff $c \in class(r)$. That is, the classes that r belongs to must be the values of r's `objectClass` attribute.

(d) $dn : R \to \texttt{distingushedName}$ is a function that associates with each directory entry r a sequence $s_1, \ldots, s_n$ of sets of (attribute, value) pairs, referred to as the *distinguished name* of r. The first set, s_1, in the sequence is called the *relative distinguished name* of r, denoted by $rdn(r)$. Distinguished names must satisfy the conditions: (i) $\forall r, r' \in R : r \neq r' \Rightarrow dn(r) \neq dn(r')$, that is, dn must be a key of each directory entry; and (ii) $rdn(r) \subseteq val(r)$. ∎

We use the (relative) distinguished names to induce a hierarchy among directory entries. We say that: (a) entry $r \in R$ is a *parent* of entry $r' \in R$ if $dn(r') = rdn(r'), dn(r)$; entry r' is said to be a *child* of entry r. (b) entry $r \in R$ is an *ancestor* of entry $r' \in R$ if there exist sets of (attribute, value) pairs $s_1, \ldots, s_m$ s.t. $dn(r') = s_1, \ldots, s_m, dn(r)$; entry r' is said to be a *descendant* of r. Abusing terminology, we also use the hierarchical relationships between distinguished names, e.g., to say that $dn(r)$ is a parent (child, ancestor, descendant) of $dn(r')$.

Note the resemblance between (relative) distinguished names and fully qualified file names in the UNIX system. While the later uses only one attribute (the file name) to distinguish between the files in a UNIX directory, our model allows more flexibility using an arbitrary set of (attribute, value) pairs to distinguish between the children of a directory entry. Also observe that since $\texttt{distinguishedName} \in \mathcal{T}$, directory entries can have attributes whose value is the dn of some other directory entry and, hence, can serve as directory entry references. This is further discussed in the sequel.

Directory entries are the basic units for holding information in the directory data model, similar to records in the relational model, and objects in the object-oriented model. A significant difference arises, as we shall see shortly, due to the modeling flexibility allowed by the directory data model. Several examples of directory instances and the use of distinguished names are presented below in Section 3.4.

3.3 Hierarchical Directory Namespace and the Influence of DNS

Each directory entry r is associated with a unique name, its distinguished name, and the set of entries is organized into a hierarchical namespace; this hierarchical organization is called the *directory information forest* (DIF).[3] The hierarchical directory namespace typically corresponds to administrative responsibilities for portions of the namespace, and may reflect political, geographic, and/or organizational boundaries. Different network operators or large businesses own portions of the namespace and operate their own directory servers for their part of the namespace. This is very similar to the way the Domain Name System (DNS) operates, which has served superlatively in allowing maintenance of its namespace in a distributed fashion, and in providing very rapid lookups in the namespace [24].

[3] In LDAP, this is referred to as a *directory information tree*, but in the formal model we develop, this could be a forest. We need this extension to obtain the closure property for our query languages.

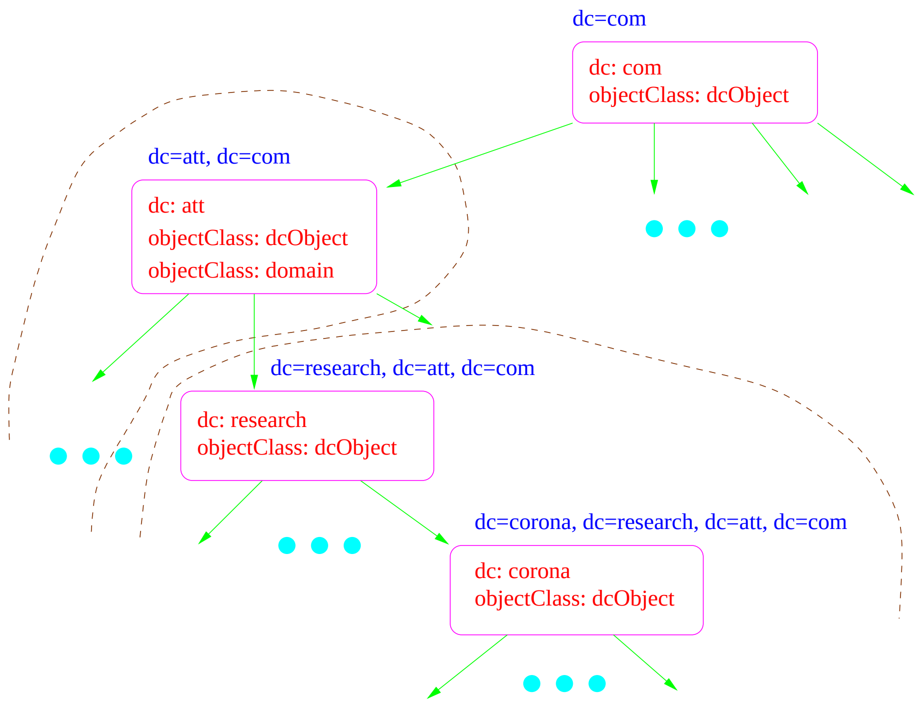

Figure 1: Higher Levels of the Network Directory Information Forest

As with DNS, the network directory can be maintained in a highly distributed fashion, with each directory server providing directory services for a limited number of "domains" in the directory information forest. The basic mechanism is akin to DNS in that at the time of registration of a domain in the DIF, a primary and (perhaps) some secondary directory servers[4] are identified as the owners of the hierarchical namespace rooted at the domain entry. Each of these directory servers must provide directory services for each "host" in the domain. As with DNS, it is also possible to split a domain into subdomains, with a different (primary and secondary) directory server for each subdomain. Thus, the network directory service can be supplied in a highly distributed fashion.

3.4 Motivating Applications Revisited

For our motivating network applications, the higher levels of the directory information forest correspond to the DNS domain and host name hierarchy. Figure 1 illustrates a portion of the higher levels of the DIF, using the attributes `dc` (or `domainComponent`) and `objectClass`, and classes `domain` and `dcObject` from the Netscape Directory Server 3.1. The dotted lines demarcate the domain and subdomain boundaries between different directory servers.

The distinguished name of each directory entry is indicated above the box surrounding it in the figure. For simplicity, the *rdn*'s in all our examples contain a single (attribute, value) pair. Hence, instead of writing a *dn* as a sequence of singleton sets of (attribute, value) pairs, we simply write the *dn* as a sequence of (attribute, value) pairs.

[4] Secondary directory servers ensure that one unreachable network will not necessarily cut off network directory service.

Example 3.1 [Supporting Quality of Service (QoS) in the Core Network]
Figure 12, in the appendix, depicts sample data useful for supporting different qualities of service within the `research.att.com` subnet, based on the schema descriptions in [11]. Four types of directory entries needed for service level administration are depicted in the figure: `SLAPolicyRules` entries (each entry represents a distinct policy), `trafficProfile` entries (each entry represents a distinct network packet profile), `policyValidityPeriod` entries (each entry represents information about the temporal applicability of the policy), and `SLADSAction` entries (each entry represents a distinct action to be performed).

The policy with *dn* `SLAPolicyName=dso, ou=SLAPolicyRules, ou=networkPolicies, dc=research, dc=att, dc=com` applies to data traffic whose IP source address matches `204.178.16.*` or `207.140.*.*`, during weekends and Thanksgiving in 1998. All packets that match this profile are denied permission into the subnet. The policy's traffic profiles are described in two separate entries with *dn* `TPName=lsplitOff, ou=trafficProfile, ou=networkPolicies, dc=research, dc=att, dc=com` and *dn* `TPName=csplitOff, ou=trafficProfile, ou=networkPolicies, dc=research, dc=att, dc=com`; only the first entry is depicted in the figure. The validity periods of the policy are also specified in two distinct directory entries with the *dn*'s `PVPName=1998weekend, ou=policyValidityPeriod, ou=networkPolicies, dc=research, dc=att, dc=com`, and `PVPName=1998thanksgiving, ou=policyValidityPeriod, ou=networkPolicies, dc=research, dc=att, dc=com`; only the first entry is depicted in the figure. The start time and end time in these entries follow the year-month-date-hour-minute-second format. Finally, the action indicated by the policy is specified in a separate directory entry with *dn* `DSActionName=denyAll, ou=SLADSAction, ou=networkPolicies, dc=research, dc=att, dc=com`. This policy is specified to have two exceptions, each of which is itself a policy below `ou=SLAPolicyRules, ou=networkPolicies, dc=research, dc=att, dc=com`, and is not shown in the figure for lack of space. ∎

Example 3.2 [Supporting Location and Device Independent Access]
The TOPS application currently stores its subscriber data and the query handling profiles in a home-grown directory, customized for its needs [2, 3]. Figure 11, in the appendix, depicts some sample TOPS data modeled in the LDAP-based network directory data model. Each TOPS subscriber is associated with a subtree whose root is a child of the directory entry with *dn* `ou=userProfiles, dc=research, dc=att, dc=com`.

The root of such a subtree is a directory entry with the profile of the TOPS subscriber, shown in the figure as having classes `inetOrgPerson` and `TOPSSubscriber`, and additionally specifying values for attributes `surName`, `commonName` and `uid`. The various TOPS query handling profiles (QHPs) are children entries of the TOPS subscriber. Two sample entries are depicted in the figure: Jagadish's weekend QHP, which has a higher priority (a lower value for the `priority` attribute), and Jagadish's working hours QHP, which has a lower priority. Each of the call appearances corresponding to a given QHP is a child entry of the QHP entry in the directory. Two sample call appearance entries are depicted in the figure corresponding to Jagadish's working hours QHP: his office phone number, which has a higher priority; and his secretary's office phone number, which has a lower priority; his voice messaging mailbox appears (not shown) with a still lower priority. On the other hand, his voice messaging mailbox may be the only call appearance specified corresponding to his weekend QHP.

A key difference between the nature of the data modeling in the QoS application and in the TOPS application is that the hierarchical directory namespace in the QoS application is partitioned based on functionality (profiles, policies, actions), whereas the hierarchical directory namespace in the TOPS application is partitioned by subscriber. Thus, different TOPS subscribers own non-overlapping portions of the hierarchical directory namespace, and each TOPS subscriber represents and manages his own policies, profiles and actions in his personal namespace; this is ideal for TOPS, and similar *personal directory* applications. ▮

3.5 Naturalness of the Network Directory Data Model

One may wonder whether directories are indeed natural for storing the data in our motivating applications. Why not a relational or an object-oriented database? There are two significant reasons why directories are more appropriate here than relational or object-oriented databases.

First, the directory data model defines a hierarchical namespace for entries which enables highly distributed management of entries across directory servers in the network, while still permitting a conceptually unified view of the data. This is not directly supported by the relational and object-oriented models.

Second, different subnets in the network may define their profiles and policies in very different ways, reflecting the heterogeneity in real-world entities. The directory data model can represent and manipulate such heterogeneity in a very easy, natural and flexible manner, which is critical for ensuring the autonomy of the different directory servers:

- *A directory entry can belong to multiple classes, and different entries may belong to very different sets of classes.* Thus, an entry can specify values for attributes in the definitions of any of its classes, without requiring a single class to contain this union of attributes in its definition.

  For example, some user may belong to classes `inetOrgPerson` and `TOPSSubscriber` (see Example 3.2), while another user may belong to classes `inetOrgPerson` and `ntUser`. Neither do these classes (all except `TOPSSubscriber` are taken from the default schema of Netscape Directory Server 3.1) need to be pairwise related by any class-subclass relationship in the directory schema, nor is there a need to define a subclass that inherits from, e.g., both `inetOrgPerson` and `TOPSSubscriber`.

- *Different entries belonging to the same set of classes may contain very different attributes.* The semantics of attributes in class definitions allows for this form of heterogeneity.

  For example, some query handling profile for a TOPS subscriber may specify values for the `startTime` and `endTime` attributes, another query handling profile may specify values for the `daysOfWeek` attribute, while a third query handling profile may not specify a value for any of these attributes.

- *A directory entry can have multiple values for an attribute.* This allows for some heterogeneity among directory entries that have exactly the same set of attributes.

  For example, a network policy rule (see Example 3.1) may specify multiple values for each of its `SLAExceptionRef`, `SLATPRef` and `SLAPVPRef` attributes. Similarly, a `policyValidityPeriod` entry may specify multiple values for its `PVDayOfWeek` attribute.

In comparison, the relational and object-oriented models are considerably more rigid for our applications. A record in the relational model can belong to just one table and an attribute can have only a single value. An object in the object-oriented data model can belong to just one most-specific class, and objects (of the same most-specific class) are very homogeneous.

The directory data model shares the flexibility of the recently proposed models for semi-structured data (see, e.g., [1, 8]), while at the same time effectively addressing the specific needs of network directory applications. The specific "restrictions" we impose, such as the hierarchical namespace, are typical in the context of network directories, and critical for performance, as discussed above. However, arbitrary DAGs and cyclic data can easily be described by having attributes "pointing" to the referenced entries. Such references are also supported by the query language, as explained in Section 7.

4 The Query Language $\mathcal{L}_0$: Boolean Operators

4.1 Atomic Queries

An *atomic query* consists of a base directory entry, a search scope, and an atomic filter, similar to an atomic LDAP query [30, 17, 18]. The *base entry*, specified by its distinguished name, is the entry relative to which the filter is to be evaluated. The *scope* indicates whether the filter is to be evaluated only at the base entry (`base`), down to all children of the base entry (`one`), or down to all descendants of the base entry (`sub`).

The choice of atomic filters depends on the set of base types, $\mathcal{T}$, in the directory data model. For concreteness, we use atomic filters for the base types `string` and `int` in our examples. These atomic filters can compare individual attributes with integer values (e.g., `SLARulePriority < 3`), test for the presence of an attribute (e.g., `telephoneNumber=*`), or do wildcard comparisons with the string value of an attribute (e.g., `commonName=*jag*`).

Intuitively, a directory entry satisfies an atomic filter if *at least one* of the (attribute, value) pairs of the entry satisfies the filter. Below, we define what it means for a directory entry r to satisfy an atomic filter F, denoted $r \models F$, for a few representative atomic filters.

$$
\begin{aligned}
r \models a = * &\stackrel{\text{def}}{=} \exists(v)((a, v) \in val(r)) \\
r \models a < v_1 &\stackrel{\text{def}}{=} \exists(v_2)(\tau(a) = \texttt{int} \wedge (a, v_2) \in val(r) \wedge v_2 < v_1) \\
r \models a = *v_2* &\stackrel{\text{def}}{=} \exists(v, v_1, v_3)(\tau(a) = \texttt{string} \wedge (a, v) \in val(r) \wedge v = v_1 v_2 v_3)
\end{aligned}
$$

In general, a query Q is a function that maps a directory instance $I = (R, class, val, dn)$ of directory schema S to an instance $I' = (R', class, val, dn)$ of schema S, such that $R' \subseteq R$. Since all other components remain unchanged, we only enumerate the result set of directory entries R' when specifying the semantics of a query Q, denoted by $\mathcal{M}(Q)$.

Definition 4.1 (Semantics of an Atomic Query) The semantics of an atomic query $(B \ ? \ Scope \ ? \ F)$, is given by enumerating the possible values of the scope *Scope*:

$$
\begin{aligned}
\mathcal{M}(B \ ? \ \texttt{base} \ ? \ F) &\stackrel{\text{def}}{=} \{r \mid r \in R \wedge r \models F \wedge dn(r) = B\} \\
\mathcal{M}(B \ ? \ \texttt{one} \ ? \ F) &\stackrel{\text{def}}{=} \{r \mid r \in R \wedge r \models F \wedge (dn(r) = B \vee dn(r) \text{ is a child of } B)\} \\
\mathcal{M}(B \ ? \ \texttt{sub} \ ? \ F) &\stackrel{\text{def}}{=} \{r \mid r \in R \wedge r \models F \wedge (dn(r) = B \vee dn(r) \text{ is a descendant of } B)\} \ \blacksquare
\end{aligned}
$$

We require that atomic queries can be evaluated efficiently. This is not a major assumption, since the atomic queries considered above are all supported by LDAP, and can be evaluated with the help of B-trees indices for integer and distinguishedName filters, and trie and suffix tree indices [23] for string filters.

4.2 Boolean Operators

Atomic queries can be combined using the boolean operators: **and** (&), **or** (|), and **set difference** (−), to form complex $\mathcal{L}_0$ queries, using the grammar in Figure 7 in the appendix. The boolean operators have the obvious set-theoretic semantics.

Observe that, in LDAP, only atomic filters (but not queries) can be combined using the boolean operators: **and** (&), **or** (|), **not** (!), to form complex LDAP filters. That is, a complex LDAP query can have a single base-entry-DN and a single scope, whereas different atomic queries in a complex $\mathcal{L}_0$ query may have different base-entry-DNs and different scopes. We have not defined the LDAP query language formally, since it is virtually identical, for our purposes, to $\mathcal{L}_0$, except for this one material difference.

Example 4.1 [Different Base Entries Can be Useful]
To locate directory entries whose `surName` is `jagadish` in AT&T, except those in Research, we can formulate the following $\mathcal{L}_0$ query:

```
(−  (dc=att, dc=com ? sub ? surName=jagadish)
    (dc=research, dc=att, dc=com ? sub ? surName=jagadish))
```

This query cannot be formulated as a single LDAP query (see Section 8); the application would have to pose two separate LDAP queries, and compute the difference within the application. ∎

A boolean expression can be evaluated efficiently, using straightforward list merging techniques, when each of the inputs to the boolean operator is represented as a sorted list. Jacobson et al [21] describe an elegant table-driven algorithm for this task. Given sorted lists L_1, L_2, the results of each of the boolean queries "(& L_1 L_2)", "(| L_1 L_2)" and "(− L_1 L_2)" can be computed with linear I/O complexity by scanning the input lists once in sorted order, and writing out the output list. The result of the boolean query, computed using these list merging techniques, is also sorted.

For reasons that will become obvious when we consider the evaluation of more sophisticated queries over the directory, we choose the sort order to be *the lexicographic ordering on the reverse of the string representation of the distinguished names of the directory entries* [31].

5 The Query Language $\mathcal{L}_1$: Hierarchy Operators

Queries expressed in $\mathcal{L}_0$ over a network directory, while richer than standard LDAP queries, can take advantage of the hierarchical organization of the directory entries in only a very limited fashion, as we see in the examples below. The language $\mathcal{L}_1$ extends $\mathcal{L}_0$ with *hierarchical selection* operators that allow much richer means for exploiting the hierarchical organization of directory entries. We present examples before describing the formal semantics of the additional operators.

5.1 Illustrative Examples

Example 5.1 [Selecting Parents and Children]
Suppose we want to ask the query "Find organizational units in AT&T that directly contain entries with

surName=jagadish". All organizational units can be located using the $\mathcal{L}_0$ query "(dc=att, dc=com ? sub ? objectClass=organizationalUnit)". All entries in AT&T with surName=jagadish can be located using the $\mathcal{L}_0$ query "(dc=att, dc=com ? sub ? surName=jagadish)". These two $\mathcal{L}_0$ queries can be composed into a single $\mathcal{L}_1$ query, to obtain the desired result, using the binary hierarchical selection operator children (c), as follows:

```
(c   (dc=att, dc=com ? sub ? objectClass=organizationalUnit)
     (dc=att, dc=com ? sub ? surName=jagadish))
```

This query returns each entry that satisfies the first operand of the binary children operator: "(dc=att, dc=com ? sub ? objectClass=organizationalUnit)", and has at least one child entry that satisfies the second operand of the binary children operator: "(dc=att, dc=com ? sub ? surName=jagadish)".

Similar examples also arise when trying to model and unambiguously locate organizational and personal lists in directories; see [22] for more details. ∎

Example 5.2 [Selecting Ancestors and Descendants]
Suppose we want to ask the query "Find the various traffic profiles useful for the network policies governing the various subnets of dc=att, dc=com". From the hierarchical organization of directory entries, we know that all such entries are in the subtree rooted at the directory entry dc=att, dc=com. However, formulating the obvious $\mathcal{L}_0$ query "(dc=att, dc=com ? sub ? objectClass=trafficProfile)" could additionally retrieve traffic profiles that are represented in the directory but not used in network policies. All and only the desired traffic profiles are in the subtrees rooted at entries with ou=networkPolicies. All such directory entries can be located using the $\mathcal{L}_0$ query "(dc=att, dc=com ? sub ? ou=networkPolicies)". These two $\mathcal{L}_0$ queries can be composed into a single $\mathcal{L}_1$ query, to obtain the desired result, using the binary hierarchical selection operator ancestors (a), as follows:

```
(a   (dc=att, dc=com ? sub ? objectClass=trafficProfile)
     (dc=att, dc=com ? sub ? ou=networkPolicies))
```

This query returns each directory entry that satisfies the first operand of the binary ancestors operator: "(dc=att, dc=com ? sub ? objectClass=trafficProfile)" and has at least one (proper) ancestor directory entry that satisfies the second operand of the binary ancestors operator "(dc=att, dc=com ? sub ? ou=networkPolicies)". ∎

Example 5.3 [Selecting Path Constrained Ancestors and Descendants]
The hierarchical organization of directory entries depicted in Figures 1, 12 and 11 illustrate that often dcObject entries are children of other dcObject entries, and organizationalUnit entries are children of other organizationalUnit entries. One often wishes to locate the *closest* dcObject ancestor entry of another directory entry.[5] Consider, for example, the query "Which subnets in AT&T have specified traffic profiles governing SMTP traffic (that is, from port 25)"? The above query can, for example, be specified as:

(d_c (dc=att, dc=com ? sub ? objectClass=dcObject)

[5] Using the descendants operator, one could locate *all* the dcObject ancestor entries.

```
(&  (dc=att, dc=com ? sub ? sourcePort=25)
    (dc=att, dc=com ? sub ? objectClass=trafficProfile))
(dc=att, dc=com ? sub ? objectClass=dcObject))
```

This query returns each entry r_1 that satisfies the first operand of the ternary $\texttt{descendants}_c$ operator: "`(dc=att, dc=com ? sub ? objectClass=dcObject)`", and has at least one descendant entry r_2 that satisfies the second operand of the $\texttt{descendants}_c$ operator: "`(& (dc=att, dc=com ? sub ? sourcePort=25) (dc=att, dc=com ? sub ? objectClass=trafficProfile))`", provided there is no entry $r_3, r_3 \neq r_1, r_3 \neq r_2$, such that r_3 is a descendant entry of r_1 and r_2 is a descendant entry of r_3, and r_3 satisfies the third operand of the $\texttt{descendants}_c$ operator: "`(dc=att, dc=com ? sub ? objectClass=dcObject)`". ∎

5.2 Hierarchical Selection Formalized

As explained above, $\mathcal{L}_1$ extends $\mathcal{L}_0$ with six operators, and is defined by the extension, presented in Figure 8 in the appendix, to the grammar of $\mathcal{L}_0$.

Definition 5.1 (Semantics of Hierarchical Selection Queries) We define the semantics of the six hierarchical selection operators below:

$$
\begin{aligned}
\mathcal{M}(c\ Q_1 Q_2) &\stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge\ r_2 \text{ is a child of } r_1))\} \\
\mathcal{M}(p\ Q_1 Q_2) &\stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge\ r_2 \text{ is a parent of } r_1))\} \\
\mathcal{M}(d\ Q_1 Q_2) &\stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge\ r_2 \text{ is a descendant of } r_1))\} \\
\mathcal{M}(a\ Q_1 Q_2) &\stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge\ r_2 \text{ is an ancestor of } r_1))\} \\
\mathcal{M}(d_c\ Q_1 Q_2 Q_3) &\stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge r_2 \text{ is a descendant of } r_1 \wedge \\
&\qquad (\nexists(r_3)(r_3 \in \mathcal{M}(Q_3) \wedge r_3 \text{ is a descendant of } r_1 \wedge\ r_2 \text{ is a descendant of } r_3))))\} \\
\mathcal{M}(a_c\ Q_1 Q_2 Q_3) &\stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge r_2 \text{ is an ancestor of } r_1 \wedge \\
&\qquad (\nexists(r_3)(r_3 \in \mathcal{M}(Q_3) \wedge r_3 \text{ is an ancestor of } r_1 \wedge\ r_2 \text{ is an ancestor of } r_3))))\}
\end{aligned}
$$
∎

5.3 Evaluating Hierarchical Selection Operators

The straightforward way of computing the hierarchical selection operators, `parents` and `children`, by *independently* testing whether each entry of the first operand is in the output by finding a "witness" entry in the second operand, is quadratic in the sum of the sizes of the two operands.

The key to a more efficient computation of `parents` and `children` is to use a stack-based algorithm in conjunction with a sorted representation of the two operands using lexicographic ordering of the reverse of the dn's. Such an algorithm (with efficient CPU time complexity) was presented in [22]. In this paper, we have adapted their algorithm to (a) improve the I/O complexity of the computation, and (b) allow algorithms for the other operators to be obtained by judicious modification of this basic algorithm. This modified algorithm, Algorithm ComputeHSPC, is presented in Figure 2.

The algorithm works for precisely the same reasons as the algorithm presented in [22], and we paraphrase their argument below because it will help us in understanding the correctness of subsequent algorithms in this

Algorithm ComputeHSPC (op, L_1, L_2) {

  Assumption: each of L_1 and L_2 are sorted based on the lexicographic ordering of the reverse dn's.

    /* the reverse dn of a parent entry is a prefix of the reverse dn of a child entry */

  /* Phase 1: each entry in L_1 is associated with the number of its parents and children in L_2 */

  Initially stack S is empty, entry r_l = firstElement(L_1, L_2), and label$(r_l) = \{i \mid r_l \in L_i\}$.

    /* r_l points to the first entry in the lexicographic merge of L_1 and L_2 */

  repeat

    below$(r_l) = 0$, above$(r_l) = 0$.

    if (stack S is empty) push r_l on top of stack S, and r_l = nextElement(L_1, L_2).

    else

      let r_t be the entry at the top of the stack S.

      if (r_t is an ancestor of r_l)

        if (($2 \in$ label(r_l)) and (r_t is a parent of r_l)) above(r_t) = above$(r_t)+1$.

        if (($2 \in$ label(r_t)) and (r_t is a parent of r_l)) below$(r_l) = 1$.

        push r_l on top of stack S, and r_l = nextElement(L_1, L_2).

      else

        if ($1 \in$ label(r_t)) associate values (above(r_t), below(r_t)) with entry r_t in list L_1.

        pop stack.

  until (all entries in L_1 and L_2 have been processed and stack S is empty).

  /* Phase 2: list L_1 is scanned in order, and the result is output */

  entry r_l = firstElement(L_1).

  repeat

    if ((op is p) and (below$(r_l) > 0$)) output r_l.

    else if ((op is c) and (above$(r_l) > 0$)) output r_l.

    r_l = nextElement(L_1).

  until (all entries in L_1 have been processed).

}

Figure 2: Efficiently Computing `parents` and `children`

paper. The correctness of Algorithm ComputeHSPC is based on the following two observations. (1) Adjacent entries on the stack always correspond to immediate (that is, no intervening entries) ancestor/descendant pairs in the directory, from among the entries in the merge of lists L_1 and L_2. Also, every immediate ancestor/descendant pair in the merge of lists L_1 and L_2 will be adjacent to each other on the stack at some point. (2) When an entry is pushed to the top of the stack, all its ancestors in the merge of lists L_1 and L_2 are present on the stack. Also, an entry is removed from the stack only after all its descendants in the merge of L_1 and L_2 have been removed from the stack. Finally, the output of Algorithm ComputeHSPC is also sorted in the lexicographic order of reverse dn's.

The stack-based Algorithm ComputeHSPC can be extended to compute the hierarchical selection operators `ancestors` and `descendants`, as well as the path constrained hierarchical selection operators $\texttt{ancestors}_c$ and $\texttt{descendants}_c$.

For computing `ancestors` and `descendants`, we maintain two counts with each entry on the stack: (a) the number of lower (ancestor) stack entries belonging to list L_2, and (b) the number of higher (descendant) stack entries belonging to list L_2 that were encountered. These two counts can be maintained in an incremental fashion, when entries are pushed onto or popped from the stack. Doing so results in a stack-based algorithm for computing `ancestors` and `descendants`. Algorithm ComputeHSAD in Figure 4 in the appendix presents the detailed algorithm; the differences from Algorithm ComputeHSPC in Figure 2 are highlighted in bold.

For computing $\texttt{ancestors}_c$ and $\texttt{descendants}_c$, the key is to keep track of stack entries from list L_3, and not to propagate the above(r_t) and below(r_t) counts *through* stack entries that are from L_3. These two counts can be maintained in an incremental fashion, when entries are pushed onto or popped from the stack. Doing so results in a stack-based algorithm for computing $\texttt{ancestors}_c$ and $\texttt{descendants}_c$. Algorithm ComputeHSAD$_c$ in Figure 5 in the appendix presents the detailed algorithm; again, the differences from Algorithm ComputeHSAD in Figure 4 are highlighted in bold.

A careful analysis of the algorithms show that all three have linear I/O complexity. The crux of the proof is the observation that although particular stack entries may be swapped out (and eventually re-fetched) from the memory multiple times when the stack repeatedly grows and shrinks, the overall I/O that the processing performs is either $O(\frac{|L_1|}{B} + \frac{|L_2|}{B})$ or $O(\frac{|L_1|}{B} + \frac{|L_2|}{B} + \frac{|L_3|}{B})$, depending on the algorithm, where B is the blocking factor, that is, the number of directory entries per disk page.

Theorem 5.1 *Algorithm ComputeHSPC correctly computes (p L_1 L_2) and (c L_1 L_2), Algorithm ComputeHSAD correctly computes (a L_1 L_2) and (d L_1 L_2), and Algorithm ComputeHSAD$_c$ correctly computes (a_c L_1 L_2 L_3) and (d_c L_1 L_2 L_3).*

Further, the I/O complexities of Algorithms ComputeHSPC and ComputeHSAD are $O(\frac{|L_1|}{B} + \frac{|L_2|}{B})$, and the I/O complexity of Algorithm ComputeHSAD is $O(\frac{|L_1|}{B} + \frac{|L_2|}{B} + \frac{|L_3|}{B})$, where B is the blocking factor. ▮

6 The Query Language $\mathcal{L}_2$: Aggregate Selection Operators

Although more powerful than $\mathcal{L}_0$, there are still some important queries not supported by $\mathcal{L}_1$. Consider the problem of identifying the highest priority policy rules specified for the management of a given subnet, or the problem of locating TOPS subscribers who have specified more than 10 query handling profiles. The naturalness of these queries in DEN applications, and the important role played by aggregation in database query languages such as SQL, suggests the desirability of being able to express such queries involving aggregation against network directories.

Introducing aggregation in our language requires considerable care. The standard approach used in relational query languages gives primacy to *aggregate computation*. Incorporating aggregate computation directly in the network directory model would potentially require the ability to dynamically create new directory entries, associate the newly computed values with attributes of these entries, and place the entries in the hierarchical namespace. Doing so would destroy the simplicity of our family of query languages, which simply select directory entries from the input directory instance. Alternatively, associating newly computed (attribute, value) pairs with *existing* directory entries would mix up the query language with the update language, and result in state-based computation, which is undesirable. Therefore, we argue that *aggregate selection* should be viewed as a primitive in itself and incorporated in our query languages.

We extend the language $\mathcal{L}_1$ to support aggregate selection in two distinct ways. First, an aggregate computation followed by a selection can be performed on the result of any (atomic or complex) query, using the `simple aggregate selection` (g) operator. A second way of supporting aggregate selection is to extend each of the hierarchical selection operators by adding an extra aggregate selection filter operand. This performs an aggregate computation followed by a selection on the *relationship* between a directory entry in the first operand of the operator, and the set of its "witnesses" in the second operand of the operator. We present examples of both before describing the formal semantics.

6.1 Illustrative Examples

Example 6.1 [Simple Aggregate Selection]
The query "Find the policy rules specified for the management of the `dc=research, dc=att, dc=com` subnet that have more than one policy validity period" can be expressed as follows:

```
(g  (dc=research, dc=att, dc=com ? sub ? objectClass=SLAPolicyRules)
    count(SLAPVPRef) > 1)
```

The aggregate selection filter "`count(SLAPVPRef) > 1`" is applied to each of the directory entries that satisfy "`(dc=research, dc=att, dc=com ? sub ? objectClass=SLAPolicyRules)`". The aggregate term `count(SLAPVPRef)` is used to associate each directory entry with a single value that is the number of values of the `SLAPVPRef` attribute in the entry. Only entries with associated value greater than 1 are returned. ∎

Example 6.2 [Structural Aggregate Selection]
The query "Find TOPS subscribers in `dc=att, dc=com` who have specified more than 10 query handling profiles" can be expressed by using the aggregate selection variant of the `children` operator as follows:

```
(c  (dc=att, dc=com ? sub ? objectClass=TOPSSubscriber)
    (dc=att, dc=com ? sub ? objectClass=QHP)
    count($2) > 10)
```

With each directory entry that satisfies "`(dc=att, dc=com ? sub ? objectClass=TOPSSubscriber)`" (the first operand of the `children` operator), are associated *all* its children entries that satisfy the query "`(dc=att, dc=com ? sub ? objectClass=QHP)`" (the second operand of the `children` operators). Against each such association, the aggregate selection condition "`count($2) > 10`" is tested, and only those TOPS subscribers that have more than 10 children `QHP` entries are returned. ∎

6.2 Aggregate Selection Formalized

$\mathcal{L}_2$ is defined by the grammar presented in Figure 9 in the appendix. An aggregate selection filter is an arithmetic condition between two aggregate attributes, where each aggregate attribute is one of: (a) an integer constant, e.g., `10`, (b) an *entry aggregate* of the form `min(SLARulePriority)`, or (c) an *entry-set aggregate* of the forms `min(min(SLARulePriority))` or `count($$)`.

Definition 6.1 (Semantics of Simple Aggregate Selection Query) Given a directory entry $r \in R$, and an entry aggregate ea of the form $\texttt{agg}(a)$, the result of applying ea on r, denoted by $ea[r]$, is given by:

$$\texttt{agg}(a)[r] \stackrel{\text{def}}{=} \texttt{agg}\{\{v \mid (a, v) \in val(r)\}\}$$

where $\{\{\ldots\}\}$ denotes a multiset of values. Given a set of directory entries $R_1 \subseteq R$, and an entry-set aggregate esa of the form $\texttt{agg1}(ea)$, where ea is an entry aggregate, the result of applying esa on R_1, denoted by $esa[R_1]$, is given by:

$$\texttt{agg1}(ea)[R_1] \stackrel{\text{def}}{=} \texttt{agg1}\{\{v \mid \exists(r)(r \in R_1 \wedge v = ea[r])\}\}$$

Given a set of directory entries $R_1 \subseteq R$, and an entry-set aggregate esa of the form $\texttt{count}(\$\$)$, the result of applying esa on R_1, denoted by $esa[R_1]$, is given by:

$$\texttt{count}(\$\$)[R_1] \stackrel{\text{def}}{=} \texttt{count}\{\{r \mid r \in R_1\}\}$$

Finally, the *semantics of a simple aggregate selection query* of the form "$(g\ Q_1\ aa_1\ \theta\ aa_2)$" is:

$$\mathcal{M}(g\ Q_1\ aa_1\ \theta\ aa_2) \stackrel{\text{def}}{=} \{r \mid r \in \mathcal{M}(Q_1) \wedge aa_1[r, \mathcal{M}(Q_1)]\ \theta\ aa_2[r, \mathcal{M}(Q_1)]\}$$

where by $aa_i[r, R_1]$ we mean one of c_i (an integer constant), $ea_i[r]$, or $esa_i[R_1]$, depending on the instantiation of the aggregate attribute, and θ denotes an integer comparison operator. ∎

Definition 6.2 (Semantics of Structural Aggregate Selection Queries) Given a directory entry $r \in R$, a set of directory entries $R_s \subseteq R$, and an entry aggregate ea of the forms $\texttt{agg}(\$1.a)$, $\texttt{agg}(\$2.a)$ or $\texttt{count}(\$2)$, the result of applying ea on the (r, R_s) pair, denoted by $ea[r, R_s]$, is given by:

$$\begin{aligned}
\texttt{agg}(\$1.a)[r, R_s] &\stackrel{\text{def}}{=} \texttt{agg}\{\{v \mid (a, v) \in val(r)\}\} \\
\texttt{agg}(\$2.a)[r, R_s] &\stackrel{\text{def}}{=} \texttt{agg}\{\{v \mid \exists(r_1)(r_1 \in R_s \wedge (a, v) \in val(r_1))\}\} \\
\texttt{count}(\$2)[r, R_s] &\stackrel{\text{def}}{=} \texttt{count}\{\{r_1 \mid r_1 \in R_s\}\}
\end{aligned}$$

Given sets of directory entries $R_1, R_2 \subseteq R$, a function $f : R_1 \to 2^{R_2}$ that maps entries in R_1 to subsets of entries in R_2, and an entry-set aggregate esa of the form $\texttt{agg1}(ea)$, where ea is an entry aggregate, the result of applying esa on the (R_1, R_2, f) triple, denoted by $esa[R_1, R_2, f]$, is given by:

$$\texttt{agg1}(ea)[R_1, R_2, f] \stackrel{\text{def}}{=} \texttt{agg1}\{\{v \mid \exists(r)(r \in R_1 \wedge v = ea[r, f(r)])\}\}$$

Given sets of directory entries $R_1, R_2 \subseteq R$, a function f as above, and an entry-set aggregate esa of the form $\mathtt{count}(\$1)$, the result of applying esa on the (R_1, R_2, f) triple, denoted by $esa[R_1, R_2, f]$, is given by:

$$\mathtt{count}(\$1)[R_1, R_2] \stackrel{\text{def}}{=} \mathtt{count}\{\{r \mid r \in R_1\}\}$$

Consider a query Q of the form "(*op* Q_1 Q_2 [Q_3] AggSelFilter)". Every directory entry in $\mathcal{M}(Q_1)$ has a (possibly empty) *op-witness set* in $\mathcal{M}(Q_2)$; for example, when *op* is a, the a-witness set of an entry is the set of all its ancestors. We define the witness set function, denoted $ws_Q : \mathcal{M}(Q_1) \rightarrow 2^{\mathcal{M}(Q_2)}$, for a couple of choices of *op*, below. The other cases are similar.

$$\begin{aligned}
ws_Q(r_1) &\stackrel{\text{def}}{=} \{r_2 \mid r_2 \in \mathcal{M}(Q_2) \wedge\ r_2 \text{ is a parent of } r_1\} \ \text{ if } op \text{ is } p \\
ws_Q(r_1) &\stackrel{\text{def}}{=} \{r_2 \mid r_2 \in \mathcal{M}(Q_2) \wedge\ r_2 \text{ is a descendant of } r_1 \wedge \\
&\quad\ (\nexists r_3 \mid r_3 \in \mathcal{M}(Q_3) \wedge\ r_3 \text{ is a descendant of } r_1 \wedge r_2 \text{ is a descendant of } r_3)\} \ \text{ if } op \text{ is } d_c
\end{aligned}$$

Finally, we define the *semantics of structural aggregate selection queries* Q of the forms "($op\ Q_1\ Q_2\ aa_1\ \theta\ aa_2$)" and "($op\ Q_1\ Q_2\ Q_3\ aa_1\ \theta\ aa_2$)":

$$\begin{aligned}
\mathcal{M}(op\ Q_1\ Q_2\ [Q_3]\ aa_1\ \theta\ aa_2) \stackrel{\text{def}}{=} &\ \{r \mid r \in \mathcal{M}(Q_1) \wedge aa_1[r, ws_Q(r), \mathcal{M}(Q_1), \mathcal{M}(Q_2), ws_Q]\ \theta \\
&\ aa_2[r, ws_Q(r), \mathcal{M}(Q_1), \mathcal{M}(Q_2), ws_Q]\}
\end{aligned}$$

where by $aa_i[r, R', R_1, R_2, f]$ we mean one of c_i (an integer constant), $ea_i[r, R']$, or $esa_i[R_1, R_2, f]$, depending on the instantiation of the aggregate attribute, and θ denotes an integer comparison operator. ∎

Note that the $\mathcal{L}_1$ hierarchical selection operators are special cases of the structural aggregate selection operators, obtained by setting the aggregate selection condition to "`count($2)` $>$ `0`".

6.3 Evaluating Simple Aggregate Selection

A simple aggregate selection expression in $\mathcal{L}_2$ of the form "(g L_1 `AggSelFilter`)", where L_1 is a sorted list of directory entries, can be evaluated using at most two scans over the input list L_1.

In the first scan, individual *entry aggregates* of the form `min(SLARulePriority)` can be computed on a per-directory entry basis, and the aggregates can be associated with the directory entry itself in list L_1. During this first scan, *entry-set aggregates* of the form `count($$)` and `min(min(SLARulePriority))` can also be incrementally computed, using techniques similar to those described in Ross et al. [27], and these aggregates can be associated with the list L_1 itself. During the second scan of list L_1, a directory entry r is determined to be in the result by comparing entry aggregates associated with r, possibly with entry-set aggregates associated with L_1 or with constants, depending on the form of the `AggSelFilter`.

Theorem 6.1 *A simple aggregate selection expression in $\mathcal{L}_2$ of the form "(g L_1 `AggSelFilter`)" can be computed in $O(\frac{|L_1|}{B})$ I/O complexity, where B is the blocking factor.* ∎

6.4 Evaluating Structural Aggregate Selection

The stack-based algorithms ComputeHSPC, ComputeHSAD and ComputeHSAD$_c$ can be readily extended to incorporate structural aggregate selection. We call the resulting algorithms ComputeHSAggPC, ComputeHSAggAD, and ComputeHSAggAD$_c$ respectively, and use ComputeHSAgg to refer to the algorithm that invokes one of these three algorithms appropriately.

For illustrative purposes, we focus on extending Algorithm ComputeHSAD. As specified in Figure 4 in the appendix, ComputeHSAD first computes, for each directory entry in list L_1, the total number of its ancestors and its descendants in list L_2, and then selects directory entries based on appropriate non-zero counts. That is, the algorithm first computes the entry aggregate `count($2)`, and then checks the aggregate selection condition `count($2)` > 0. The technique of *incrementally* computing the value of the entry aggregate `count($2)`, for a directory entry r, using the values of the entry aggregates of the entries above and below entry r on stack S can be easily generalized to compute *entry aggregates* as well as *entry-set aggregates* that use the aggregate functions `min`, `max`, `sum`, `count` and `average`. In general, any "distributive" or "algebraic" aggregate [27] can be computed in this fashion. In Figure 6 in the appendix, we instantiate the general procedure for the aggregate selection filter `count($2)=max(count($2))`, that is, identifying entries in list L_1 that have the *most* ancestors/descendants in list L_2. Again, the differences from Algorithm ComputeHSAD in Figure 4 are highlighted in bold.

Theorem 6.2 *Algorithm ComputeHSAgg correctly computes (op L_1 L_2 $[L_3]$ AS), for op being one of the six hierarchical operators p, c, a, d, a_c and d_c, and AS being an aggregate selection filter. Further, the I/O complexity of Algorithm ComputeHSAgg is $O(\frac{|L_1|}{B} + \frac{|L_2|}{B}[+\frac{|L_3|}{B}])$, where B is the blocking factor.* ∎

7 The Query Language $\mathcal{L}_3$: Embedded Reference Operators

The network directory data model permits attributes to take distinguished names (dn's) as values. Since dn's can be treated as references to directory entries, such references can be embedded inside other directory entries, and relationships between directory entries can be easily modeled. For example, an `SLAPolicyRules` directory entry can be viewed as a relationship between multiple `trafficProfile` directory entries, multiple `policyValidityPeriod` directory entries, and a single `SLADSAction` directory entry.

To support such references, we extend the query language $\mathcal{L}_2$ with two symmetric operators `valueDN` (vd) and `DNvalue` (dv), that match the directory entry references embedded in a given attribute in a first list of entries with the directory entries in a second list of entries. Directory entry references that are embedded in both lists can be matched by composing these two operators. The additional grammar for queries in $\mathcal{L}_3$ (over queries in $\mathcal{L}_2$) is formally specified in Figure 10 in the appendix.

Example 7.1 [Embedded References]
The query "Find policy rules in AT&T that govern packets matching traffic profiles governing SMTP traffic" can be specified as follows:

```
(vd (dc=att, dc=com ? sub ? objectClass=SLAPolicyRules)
    (&  (dc=att, dc=com ? sub ? sourcePort=25)
```

```
        (dc=att, dc=com ? sub ? objectClass=trafficProfile))
    SLATPRef)
```

Intuitively, the semantics of the `valueDN` expression above can be understood as follows. Each directory entry that satisfies the query "`(dc=att, dc=com ? sub ? objectClass=SLAPolicyRules)`" (the first operand of the *vd* operator) is in the result of the expression, provided that a reference to at least one directory entry that satisfies the query "`(& (dc=att, dc=com ? sub ? sourcePort=25) (dc=att, dc=com ? sub ? objectClass=trafficProfile))`" (the second operand of the *vd* operator) is embedded in the `SLATPRef` attribute (the third operand of the *vd* operator) of the `SLAPolicyRules` entry.

The above query can be extended to locate the `SLADSAction` entry (that specifies the QoS action to be taken) of the the highest priority policy rule (with the smallest value of the `SLARulePriority` attribute), using the simple aggregate selection operator and the symmetric *dv* operator, as follows:

```
(dv  (dc=att, dc=com ? sub ? objectClass=SLADSAction)
     (g   (vd (dc=att, dc=com ? sub ? objectClass=SLAPolicyRules)
              (&  (dc=att, dc=com ? sub ? sourcePort=25)
                  (dc=att, dc=com ? sub ? objectClass=trafficProfile))
              SLATPRef)
          min(SLARulePriority)=min(min(SLARulePriority)))
     SLADSActRef)
```
∎

7.1 Embedded References Formalized

Definition 7.1 (Semantics of Embedded Reference Queries) The *semantics of vd and dv without aggregate selection* are as follows:

$$\mathcal{M}(vd\ Q_1\ Q_2\ a) \stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge (a, dn(r_2)) \in val(r_1)))\}$$
$$\mathcal{M}(dv\ Q_1\ Q_2\ a) \stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge (a, dn(r_1)) \in val(r_2)))\}$$

With aggregate selection, with each directory entry in $\mathcal{M}(Q_1)$, one first needs to associate the set of all its "witnesses" in $\mathcal{M}(Q_2)$. Consider an expression Q of the form "(*op* Q_1 Q_2 a `AggSelFilter`)" The definition of a witness set function is extended as follows:

$$ws_Q(r_1) \stackrel{\text{def}}{=} \{r_2 \mid r_2 \in \mathcal{M}(Q_2) \wedge (a, dn(r_2)) \in val(r_1)\} \text{ if } op \text{ is } vd$$
$$ws_Q(r_1) \stackrel{\text{def}}{=} \{r_2 \mid r_2 \in \mathcal{M}(Q_2) \wedge (a, dn(r_1)) \in val(r_2)\} \text{ if } op \text{ is } dv$$

The semantics of embedded reference queries with aggregate selection are exactly as in Definition 6.2. ∎

7.2 Evaluating Embedded References

The straightforward way of computing the embedded reference operators is to test *independently* whether each entry of the first operand is in the output by finding all its "witness" entries in the second operand. This results in a quadratic algorithm. A more efficient computation of these operators uses sort-merge

```
Algorithm ComputeERAggDV (L1, L2, A, AggSel) {
  Assumption: L1 is sorted based on the lexicographic ordering of the reverse dn's.
    /* the reverse dn of a parent entry is a prefix of the reverse dn of a child entry */
  Assumption: the aggregate selection filter AggSel is count($2)=max(count($2)).

  /* Phase 1: the embedded associations in list L2 are explicitly created in sorted order. */
  Initially the list of pairs LP is empty, and entry r_l = firstElement(L2).
  repeat
    if (r_l specifies values for attribute A)
      for each ((A, v) ∈ av(r_l)) append (v, dn(r_l)) to LP.
    r_l = nextElement(L2).
  until (all entries in L2 have been processed).
  sort LP based on the lexicographic ordering of the reverse of the dn's in the first column.

  /* Phase 2: each entry in L1 is associated with the number of its witnesses in LP,
    and the global maxima computed */
  Initially r1 = firstElement(L1), and r_p = firstElement(LP).
  Initially num(r1) = 0, and maxnum = 0.
  repeat
    if (r_p.1 = dn(r1))
      num(r1) = num(r1)+1, maxnum = max(num(r1), maxnum).
      r_p = nextElement(LP).
    else if (r_p.1 < dn(r1)) r_p = nextElement(LP).
    else associate num(r1) with entry r1 in list L1, r1 = nextElement(L1), and num(r1) = 0.
  until (all entries in L1 and LP have been processed).

  /* Phase 3: list L1 is scanned in order, and the result is output */
  entry r_l = firstElement(L1).
  repeat
    if (maxnum = num(r_l)) output r_l.
    r_l = nextElement(L1).
  until (all entries in L1 have been processed).
}
```

Figure 3: Efficiently Computing `DNvalue` with `count($2)=max(count($2))`

based techniques for join and semijoin computation from relational databases. For illustrative purposes, we focus on computing the `DNvalue` operator, for the aggregate selection filter `count($2)=max(count($2))`, in Algorithm ComputeERAggDV in Figure 3. That is, we identify entries in list L_1 that have the *most* embedded references in L_2 entries. The first step in the algorithm is to create a list of pairs LP of the associations between an entry in list L_2, and the dn's that are the values of its a attribute. Each pair in this list LP can be associated with (and is a witness for) at most one entry in list L_1. Once this list is sorted by the lexicographic ordering on the reverse of the embedded dn's, lists L_1 and LP are sorted using the same ordering, and the pairs in list LP can be *partitioned* based on the association of a pair in LP with an entry in L_1. The desired aggregate computation(s) can be performed during a single scan of each of the two sorted lists, and the aggregate selection performed in a final scan of L_1.

Other aggregate selections with the dv operator can be performed in an analogous fashion, and the computation of the symmetric vd operator can be performed similarly.

Theorem 7.1 *Algorithm ComputeERAggDV correctly computes* $(dv\ L_1\ L_2\ a\ AS)$. *Further, if* m *is the maximum number of values for attribute* a *in any entry of list* L_2*, the I/O complexity of Algorithm ComputeERAggDV is* $O(\frac{|L_1|}{B} + (\frac{|L_2|}{B} \cdot m \log(\frac{|L_2|}{B} \cdot m)))$*, where* B *is the blocking factor.* ∎

8 Comparative Assessment

8.1 Expressive Power

As we introduced each query language construct, we motivated the need for the construct and suggested why it could not be captured in the earlier languages. We formally capture this intuition in the following theorems. The proofs follow the general line of arguments previously presented, and so are omitted. By $\mathcal{LDAP}$ we mean the LDAP query language as defined in this paper. (The commercial LDAP protocol has many components beyond the query language aspects being studied here.)

Theorem 8.1 $\mathcal{LDAP} \subset \mathcal{L}_0 \subset \mathcal{L}_1 \subset \mathcal{L}_2 \subset \mathcal{L}_3$. ∎

Recall that $\mathcal{L}_1$ extends $\mathcal{L}_0$ with six new operators. We next consider the relationship between them.

Theorem 8.2 *(a)* $\mathcal{L}_0 + \{a, d\}$ *cannot express* c, p. *(b)* $\mathcal{L}_0 + \{c, p\}$ *cannot express* a, d. *(c)* $\mathcal{L}_0 + \{a, d, c, p\}$ *cannot express* a_c, d_c. *(d)* $\mathcal{L}_0 + \{a_c, d_c\}$ *can express all of* a, d, c, p. ∎

When we write $\mathcal{L}_0 + \{o_1, \ldots, o_n\}$ we mean the query language one would obtain from $\mathcal{L}_0$ by adding the operators $o_1, ..., o_n$. The above series of claims shows that a language $\mathcal{L}_0 + \{a_c, d_c\}$ has the same expressive power as the language $\mathcal{L}_1$, but with strictly fewer operators. There are two reasons for our design decision. The first is ease of use: it is much simpler to write a binary a operator than a ternary a_c operator. In addition, the way in which a_c can express `parents` is:

$(p\ Q_1\ Q_2) = (a_c\ Q_1\ Q_2$ `(null-dn ? sub ? objectClass=*)`$)$

The third argument includes the whole directory instance, and would lead to a very expensive evaluation as written, since our algorithms have I/O complexity that is linear in the size of the inputs. (Similar arguments apply to the `children` operator and d_c.)

8.2 I/O Complexity of Query Evaluation

Each query expression can be evaluated bottom-up as follows. First, the atomic queries are evaluated, and the resulting entries are sorted by the lexicographic ordering on the reverse of their *dn*'s. Next, each operator in the query tree is evaluated, as described in previous sections, and the result is *pipelined* to a higher operator in the query tree. Since each operator gets sorted input lists, and computes a sorted output list, *no additional sorting of the result of an intermediate operator is necessary to compute the query results.*

We have established, as each operator was introduced, the complexity of evaluating it using an appropriate algorithm. We can now put these results together to obtain the following theorems.

Theorem 8.3 *Any query Q in language $\mathcal{L}_2$ can be computed using constant size of main memory, with I/O complexity $O(|Q| \cdot \frac{|L|}{B})$, where $|L|$ is the cumulative size of the outputs of the atomic sub-queries of Q, $|Q|$ is the number of nodes in the query tree of Q, and B is the blocking factor.* ▮

The leaf nodes of the query tree for any query Q involve atomic selection queries, which we assume can be computed efficiently, either through a scan or using appropriate indices. This results in a cumulative L directory entries for further processing up the query tree. Every operator produces as output no more directory entries than in its inputs, and each operator can be evaluated with linear I/O complexity using a constant size memory. We can thus evaluate the query tree in reverse topologically sorted order, using constant size memory, using $|Q|$ steps, each of which in the worst case has complexity $O(\frac{|L|}{B})$.

Along the same lines, we can also establish:

Theorem 8.4 *Any query Q in language $\mathcal{L}_3$ can be computed using constant size of main memory, in time $O(|Q| \cdot \frac{|L|}{B} \cdot m \cdot log(\frac{|L|}{B} \cdot m))$, where m is the maximum number of values for any attribute in an entry.* ▮

8.3 Distributed Queries

Complex DEN queries can be issued by core network elements such as hosts, routers, firewalls, and proxy servers, as well as distributed network applications, typically to the "closest" directory server in the network. If all the data that is relevant to the query is managed by the queried directory server, then all the query processing can be performed locally at this directory server as discussed above.

In general, the data that is relevant to the query may be managed by multiple directory servers in the network. In this case, the query expression can be evaluated bottom-up as follows. First, each atomic query, whose base *dn* is managed by a directory server different from the queried server, is issued to the directory server that manages the base *dn* of the atomic query. These directory servers can be located efficiently using mechanisms similar to those used in DNS. The results of those atomic queries are shipped to the original queried directory server, which then computes the query result using the algorithms described previously.

9 Related Work

The network directory model is a hierarchical information model, which should remind readers of the early hierarchical data model (see, e.g., [28]) that led to the development of many commercial databases, notably IMS. A key difference between them is that these early DBMSs provide only navigation-based access languages for data manipulation, as opposed to declarative query languages like LDAP.

Hierarchy has been a central focus in the study of type systems (see, e.g., [10]) and description logics (see, e.g., [6]). However, in these contexts, the hierarchy is on the classes. In contrast, our hierarchy is at the instance-level, and quite orthogonal to class hierarchies. For example, two persons working for different companies may have entries that are very far apart in the forest, and yet both can be of class `organizationalPerson`.

Many algorithms are known to be very efficient over hierarchical structures. Most relevant to us in this literature are algorithms for checking the presence of sets of edges and paths. Jacobson et al. [21] present linear time merging-style algorithms for computing the elements of a list that are descendants/ancestors of some elements in a second list, in the context of focusing keyword-based searches on the Web and in UNIX-style file systems. Jagadish et al. [22] present linear time stack-based algorithms for computing elements of a list that are children/parents of some elements in a second list, in the context of supporting personal and organizational distribution lists in an LDAP directory. We build upon the works of [21] and [22], and devise stack-based and merging-style algorithms for a much larger class of queries for the directory data model.

The directory data model shares the flexibility of the graph-based models (see, e.g., GraphLog [14], Hy+ [13], and WebSQL [4]), and the recently proposed models for semi-structured data (see, e.g., Lorel [1] and UnQL [8]), while at the same time effectively addressing the specific needs of network directory applications. The specific "restrictions" we impose, such as the hierarchical namespace, are typical in the context of network directories, and are critical for distribution and performance. However, arbitrary DAGs and cyclic data can easily be described by having attributes "pointing" to the referenced entries. Graph-based and semi-structured models do not define such hierarchical namespaces, nor do they typically define node contents as sets of (attribute, value) pairs.

Several researchers have recently proposed schemas for semi-structured data, e.g., graph schemas [7], data guides [16], unary datalog schemas [25], Schema Definition Language (ScmDL) [5] and description logics [9]. The various formalisms differ in the kind of restrictions they can impose on an object's components: these vary from a simple upper bound on the sets of components (in graph schemas [7]) to arbitrary regular expressions (in ScmDL [5]). We view our approach as complementary to this body of work, and plan to investigate their integration in the future.

10 Discussion

LDAP directories have recently gained tremendous popularity. A large number of directory server implementations are now available from companies such as Critical Angle, Lucent, Netscape, Novell, Sun and Tandem. Moving out from their traditional role of network-based servers for contact information and address books, LDAP directory services are now being used in a wide variety of applications. In fact, current and near future releases of many operating systems, including Windows NT and Solaris, will have LDAP directory services to manage OS resources. LDAP directory enabled networking is being promoted by major players including AT&T, Cisco, IBM and Microsoft.

This paper represents a first attempt at devising a formal data model and sequence of query languages for this very popular "database". Our formulation has several desirable characteristics, including the closure property that permits queries to be composed, which are not obvious in current commercial LDAP systems. We have also demonstrated the inability of the current LDAP standard to express many queries needed to support applications effectively, with particular emphasis on directory-enabled networking. We have devised a series of extensions, and demonstrated that each language in this series enables the expression of a specific class of significant queries, for DEN applications, not supported by the current LDAP standard. We have shown that the increase in expressiveness is achieved without unduly increasing the computation cost.

We have implemented some of the constructs in our query language for specific directory enabled applications in AT&T, and are currently investigating their utility to other classes of network directory applications.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal on Digital Libraries*, 1(1), 1996.

[2] N. Anerousis, R. Gopalakrishnan, C. R. Kalmanek, A. E. Kaplan, W. T. Marshall, P. P. Mishra, P. Z. Onufryk, K. K. Ramakrishnan, and C. J. Sreenan. Architecture for signaling, directory services and transport for packet telephony. In *8th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Cambridge, England, July 1998.

[3] N. Anerousis, R. Gopalakrishnan, C. R. Kalmanek, A. E. Kaplan, W. T. Marshall, P. P. Mishra, P. Z. Onufryk, K. K. Ramakrishnan, and C. J. Sreenan. TOPS: An architecture for telephony over packet networks. *IEEE Journal in Selected Areas of Communication*, 1999. To appear.

[4] G. Arocena, A. Mendelzon, and G. Mihaila. Applications of a web query language. In *Proceedings of 6th International WWW Conference*, Santa Clara, CA, 1997.

[5] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Proceedings of the International Conference on Database Theory*, 1999. To appear.

[6] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. CLASSIC: A structural data model for objects. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 58–67, Portland, Oregon, June 1989.

[7] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory*, pages 336–350, Delphi, Greece, 1997.

[8] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, June 1996.

[9] D. Calvanese, G. Giacomo, and M. Lenzerini. What can knowledge representation do for semi-structured data? In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, 1998.

[10] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–521, 1986.

[11] 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.

[12] Cisco. Directory enabled networks. Available from http://www.cisco.com/warp/public/734/den/.

[13] M. Consens and A. Mendelzon. Hy$^+$: A hygraph-based query and visualization system. In *Proceedings of the ACM-SIGMOD 1993 Annual Conference on Management of Data*, pages 511–516, 1993.

[14] M. P. Consens and A. O. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proceedings of the ACM Symposium on Principles of Database Systems*, Apr. 1990.

[15] Forrester. The directory enabled network drive. Available from http://www.forrester.com/excerpts/ns980320.htm.

[16] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the International Conference on Very Large Databases*, 1997.

[17] T. Howes. The string representation of LDAP search filters. Request for Comments 2254. Available from ftp://ds.internic.net/rfc/rfc2254.txt, Dec. 1997.

[18] T. Howes and M. Smith. *LDAP: Programming directory-enabled applications with lightweight directory access protocol.* Macmillan Technical Publishing, Indianapolis, Indiana, 1997.

[19] T. Howes and M. Smith. The LDAP URL format. Request for Comments 2255. Available from ftp://ds.internic.net/rfc/rfc2255.txt, Dec. 1997.

[20] Innosoft. Innosoft's LDAP world implementation survey. Available from http://www.critical-angle.com/dir/lisurvey.html.

[21] G. Jacobson, B. Krishnamurthy, D. Srivastava, and D. Suciu. Focusing search in hierarchical structures with directory sets. In *Proceedings of the Seventh International Conference on Information and Knowledge Management (CIKM)*, Washington, DC, Nov. 1998.

[22] H. V. Jagadish, M. A. Jones, D. Srivastava, and D. Vista. Flexible list management in a directory. In *Proceedings of the Seventh International Conference on Information and Knowledge Management (CIKM)*, Washington, DC, Nov. 1998.

[23] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272, 1976.

[24] P. Mockapetris. Domain names: Concepts and facilities. Request for Comments 882. Available from ftp://ds.internic.net/rfc/rfc882.txt, 1983.

[25] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. In *Proceedings of the Workshop on Management of Semi-structured Data*, 1997.

[26] Computer Reseller News. Directory enabled network standard is near. Available from http://www.crn.com/dailies/weekending022798/feb27digJ.asp.

[27] K. A. Ross, D. Srivastava, and D. Chatziantoniou. Complex aggregation at multiple granularities. In *Proceedings of the International Conference on Extending Database Technology*, pages 263–277, 1998.

[28] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.

[29] M. Wahl, A. Coulbeck, T. Howes, and S. Kille. Lightweight directory access protocol (v3): Attribute syntax definitions. Request for Comments 2252. Available from ftp://ds.internic.net/rfc/rfc2252.txt, Dec. 1997.

[30] M. Wahl, T. Howes, and S. Kille. Lightweight directory access protocol (v3). Request for Comments 2251. Available from ftp://ds.internic.net/rfc/rfc2251.txt, Dec. 1997.

[31] M. Wahl, S. Kille, and T. Howes. Lightweight directory access protocol (v3): UTF-8 string representation of distinguished names. Request for Comments 2253. Available from ftp://ds.internic.net/rfc/rfc2253.txt, Dec. 1997.

Appendix: Algorithms, Grammars, Directory Fragments

Algorithm ComputeHSAD (op, L_1, L_2) {
  Assumption: each of L_1 and L_2 are sorted based on the lexicographic ordering of the reverse dn's.
    /* the reverse dn of a parent entry is a prefix of the reverse dn of a child entry */

  /* Phase 1: each entry in L_1 is associated with the number of its ancestors and descendants in L_2 */
  Initially stack S is empty, entry r_l = firstElement(L_1, L_2), and label$(r_l) = \{i \mid r_l \in L_i\}$.
    /* r_l points to the first entry in the lexicographic merge of L_1 and L_2 */
  repeat
    below$(r_l) = 0$, above$(r_l) = 0$.
    if (stack S is empty)
      push r_l on top of stack S, and r_l = nextElement(L_1, L_2).
    else
      let r_t be the entry at the top of the stack S.
      if (r_t is an ancestor of r_l)
        if ($2 \in$ **label**(r_l)) above(r_t) = above(r_t)+1.
        if ($2 \in$ label(r_t)) below(r_l) = below(r_t)+1.
        else below(r_l) = below(r_t).
        push r_l on top of stack S, and r_l = nextElement(L_1, L_2).
      else
        if ($1 \in$ label(r_t)) associate values (above(r_t), below(r_t)) with entry r_t in list L_1.
        pop stack.
        if (stack S is not empty)
          let r_b be the entry at the top of stack S.
          above(r_b) = above(r_b)+above(r_t).
  until (all entries in L_1 and L_2 have been processed and stack S is empty).

  /* Phase 2: list L_1 is scanned in order, and the result is output */
  entry r_l = firstElement(L_1).
  repeat
    if ((op is a) and (below$(r_l) > 0$)) output r_l.
    else if ((op is d) and (above$(r_l) > 0$)) output r_l.
    r_l = nextElement(L_1).
  until (all entries in L_1 have been processed).
}

Figure 4: Efficiently Computing `ancestors` and `descendants`

```
Algorithm ComputeHSAD_c (op, L1, L2, L3) {
  Assumption: each of L1, L2, L3 are sorted based on the lexicographic ordering of the reverse dn's.
    /* the reverse dn of a parent entry is a prefix of the reverse dn of a child entry */

  /* Phase 1: each entry in L1 is associated with the number of its ancestors and descendants in L2
    that have no intervening ancestor or descendant entry (respectively) in L3. */
  Initially stack S is empty, entry r_l = firstElement(L1, L2, L3), and label(r_l) = {i | r_l ∈ L_i}.
    /* r_l points to the first entry in the lexicographic merge of L1, L2 and L3 */
  repeat
    below(r_l) = 0, above(r_l) = 0.
    if (stack S is empty)
      push r_l on top of stack S, and r_l = nextElement(L1, L2, L3).
    else
      let r_t be the entry at the top of the stack S.
      if (r_t is an ancestor of r_l)
        if (2 ∈ label(r_l)) above(r_t) = above(r_t)+1.
        if (2 ∈ label(r_t))
          if (3 ∉ label(r_t)) below(r_l) = below(r_t)+1.
          else below(r_l) = 1.
        else if (3 ∉ label(r_t)) below(r_l) = below(r_t).
        push r_l on top of stack S, and r_l = nextElement(L1, L2, L3).
      else
        if (1 ∈ label(r_t)) associate values (above(r_t), below(r_t)) with entry r_t in list L1.
        pop stack.
        if (stack S is not empty)
          let r_b be the entry at the top of stack S.
          if (3 ∉ label(r_t)) above(r_b) = above(r_b)+above(r_t).
  until (all entries in L1 and L2 have been processed and stack S is empty).

  /* Phase 2: list L1 is scanned in order, and the result is output */
  entry r_l = firstElement(L1).
  repeat
    if ((op is a_c) and (below(r_l) > 0)) output r_l.
    else if ((op is d_c) and (above(r_l) > 0)) output r_l.
    r_l = nextElement(L1).
  until (all entries in L1 have been processed).
}
```

Figure 5: Efficiently Computing Path Constrained `ancestors`$_c$ and `descendants`$_c$

```
Algorithm ComputeHSAggAD (op, L1, L2, AggSel) {
  Assumption: each of L1 and L2 are sorted based on the lexicographic ordering of the reverse dn's.
    /* the reverse dn of a parent entry is a prefix of the reverse dn of a child entry */
  Assumption: the aggregate selection filter AggSel is count($2)=max(count($2)).

  /* Phase 1: each entry in L1 is associated with the number of its ancestors and descendants in L2,
    and the global maxima computed */
  Initially stack S is empty, entry r_l = firstElement(L1, L2), and label(r_l) = {i | r_l ∈ L_i}.
    /* r_l points to the first entry in the lexicographic merge of L1 and L2 */
  maxbelow = 0, maxabove = 0.
  repeat
    below(r_l) = 0, above(r_l) = 0.
    if (stack S is empty)
      push r_l on top of stack S, and r_l = nextElement(L1, L2).
    else
      let r_t be the entry at the top of the stack S.
      if (r_t is an ancestor of r_l)
        if (2 ∈ label(r_l)) above(r_t) = above(r_t)+1.
        if (2 ∈ label(r_t)) below(r_l) = below(r_t)+1. else below(r_l) = below(r_t).
        push r_l on top of stack S, and r_l = nextElement(L1, L2).
      else
        if (1 ∈ label(r_t))
          associate values (above(r_t), below(r_t)) with entry r_t in list L1.
          maxabove = max(above(r_t), maxabove).
          maxbelow = max(below(r_t), maxbelow).
        pop stack.
        if (stack S is not empty)
          let r_b be the entry at the top of stack S.
          above(r_b) = above(r_b)+above(r_t).
  until (all entries in L1 and L2 have been processed and stack S is empty)

  /* Phase 2: list L1 is scanned in order, and the result is output */
  entry r_l = firstElement(L1).
  repeat
    if ((op is a) and (maxbelow = below(r_l))) output r_l.
    else if ((op is d) and (maxabove = above(r_l))) output r_l.
    r_l = nextElement(L1).
  until (all entries in L1 have been processed).
}
```

Figure 6: Efficiently Computing `ancestors` and `descendants` with `count($2)=max(count($2))`

Query	:=	BooleanQuery | AtomicQuery
BooleanQuery	:=	AndQuery | OrQuery | DiffQuery
AndQuery	:=	"(&" Query Query ")"
OrQuery	:=	"(|" Query Query ")"
DiffQuery	:=	"($-$" Query Query ")"
AtomicQuery	:=	"(" BaseEntryDN "?" Scope "?" AtomicFilter ")"

Figure 7: Grammar of Query Language $\mathcal{L}_0$

Query	:=	HierarchyQuery | BooleanQuery | AtomicQuery
HierarchyQuery	:=	ParentsQuery | ChildrenQuery | AncestorsQuery | DescendantsQuery | CoAncestorsQuery | CoDescendantsQuery
ParentsQuery	:=	"(p" Query Query ")"
ChildrenQuery	:=	"(c" Query Query ")"
AncestorsQuery	:=	"(a" Query Query ")"
DescendantsQuery	:=	"(d" Query Query ")"
CoAncestorsQuery	:=	"(a_c" Query Query Query ")"
CoDescendantsQuery	:=	"(d_c" Query Query Query ")"

Figure 8: Grammar of Query Language $\mathcal{L}_1$

Query	:=	HierarchyQuery | SimpleAggQuery | BooleanQuery | AtomicQuery
SimpleAggQuery	:=	"(g" Query AggSelFilter ")"
ParentsQuery	:=	"(p Query Query [AggSelFilter] ")"
ChildrenQuery	:=	"(c Query Query [AggSelFilter] ")"
AncestorsQuery	:=	"(a Query Query [AggSelFilter] ")"
DescendantsQuery	:=	"(d Query Query [AggSelFilter] ")"
CoAncestorsQuery	:=	"(a_c Query Query Query [AggSelFilter] ")"
CoDescendantsQuery	:=	"(d_c Query Query Query [AggSelFilter] ")"
AggSelFilter	:=	AggAttribute IntOp AggAttribute
AggAttribute	:=	IntConstant | EntryAggAttr | EntrySetAggAttr
EntryAggAttr	:=	Aggregate "(" ModAttrName ")" | "count($2)"
EntrySetAggAttr	:=	Aggregate "(" EntryAggAttr ")" | "count($1)" | "count($$)"
ModAttrName	:=	AttributeName | "$1." AttributeName | "$2." AttributeName
Aggregate	:=	"max" | "min" | "count" | "sum" | "average"

Figure 9: Grammar of Query Language $\mathcal{L}_2$

Query	:=	EmbeddedRefQuery | HierarchyQuery |
		SimpleAggQuery | BooleanQuery | AtomicQuery
EmbeddedRefQuery	:=	ValueDNQuery | DNValueQuery
ValueDNQuery	:=	"(vd Query Query AttributeName [AggSelFilter] ")"
DNValueQuery	:=	"(dv Query Query AttributeName [AggSelFilter] ")"

Figure 10: Grammar of Query Language $\mathcal{L}_3$

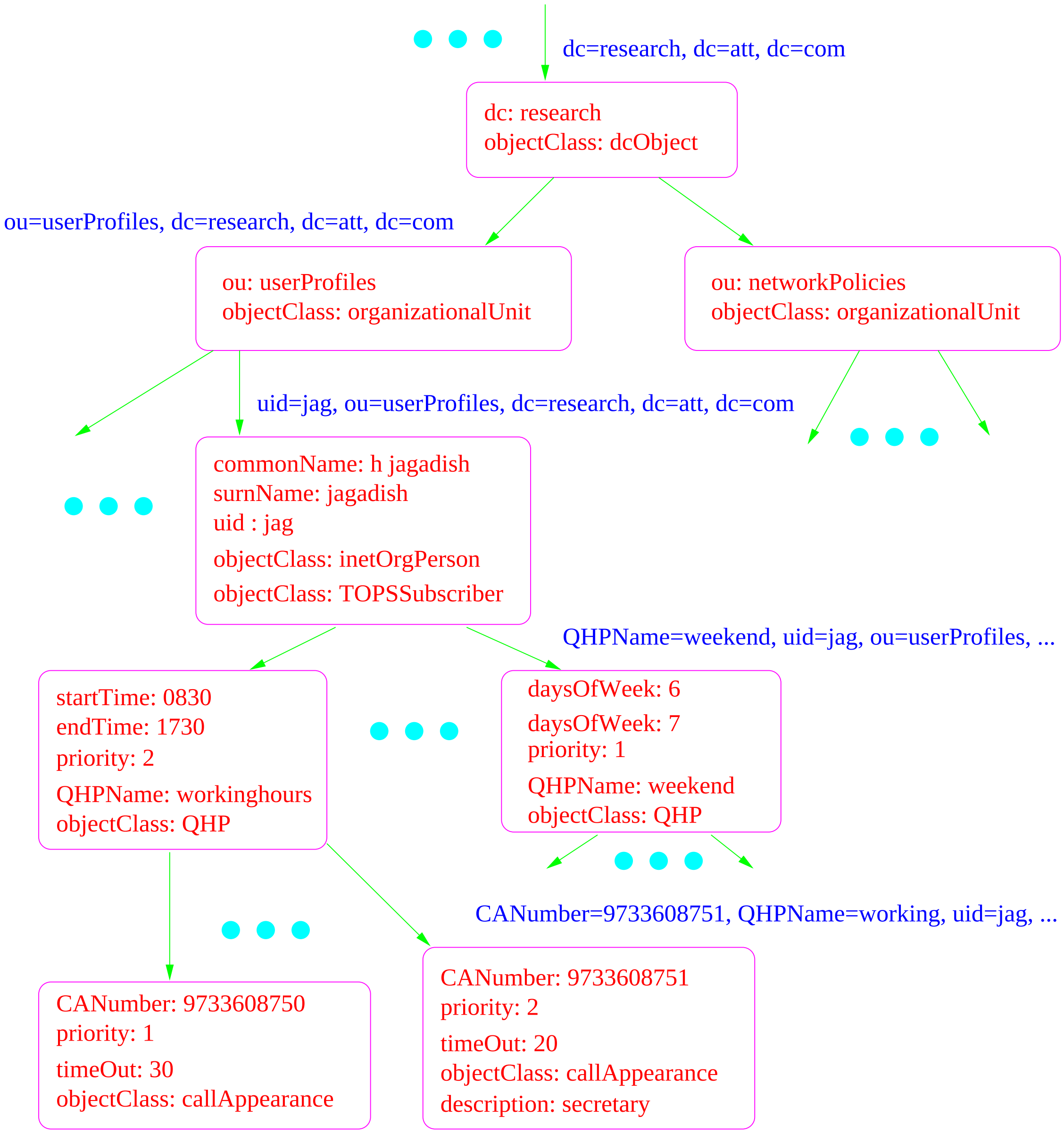

Figure 11: Directory Entries for the TOPS Application

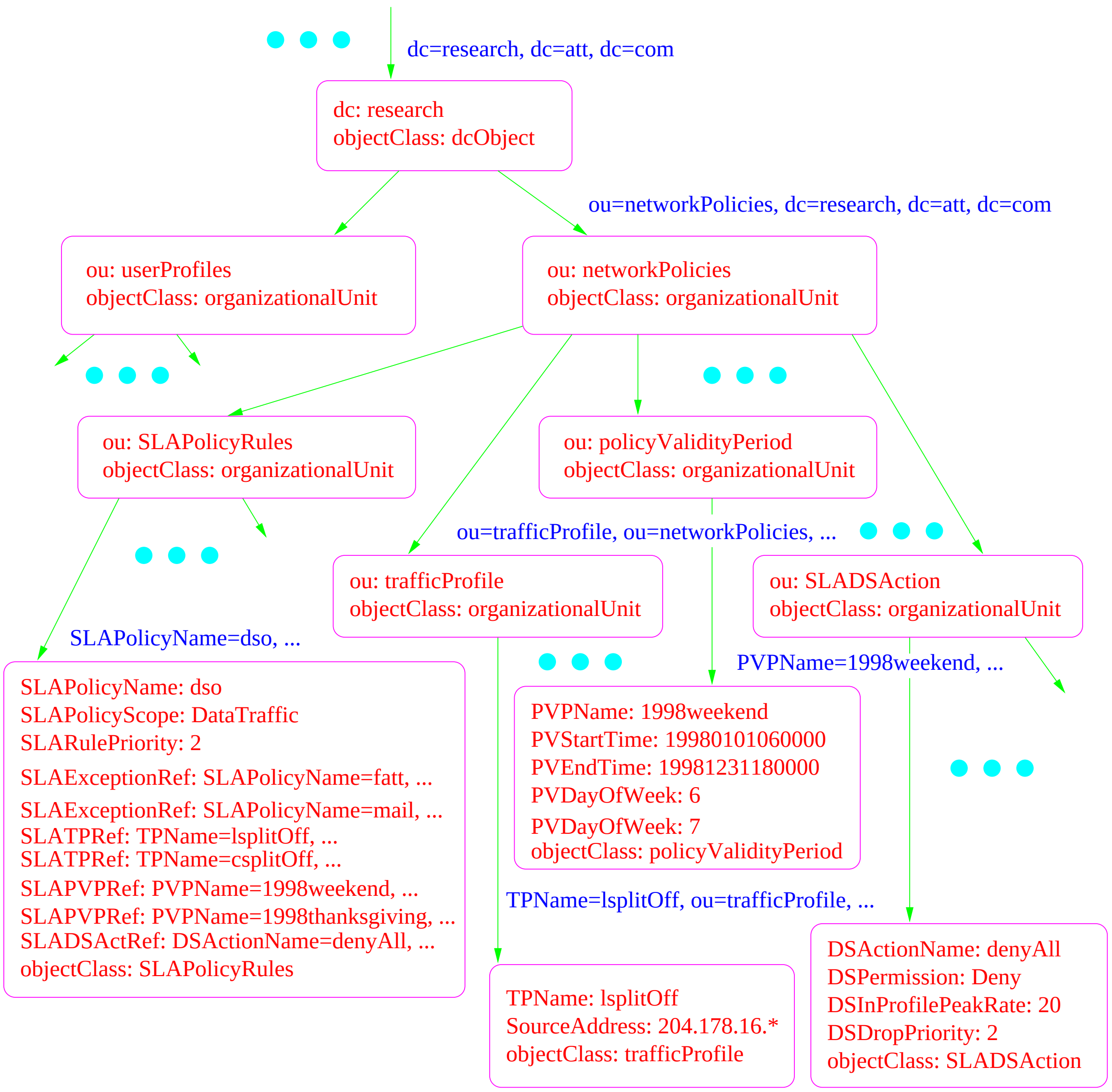

Figure 12: Directory Entries for Supporting Quality of Service in the Core Network