Querying Network Directories

H. V. Jagadish* U of Illinois, Urbana-Champaign jag@cs.uiuc.edu

Laks V. S. Lakshmanan† IIT – Bombay laks@math.iitb.ernet.in

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

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

Dimitra Vista* Drexel University dvista@mcs.drexel.edu

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" involve 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 a representative real-life example.

*This work was done when the authors were at AT&T Labs–Research, Florham Park, NJ 07932, USA.

† Currently on leave from Concordia University, Canada. This work was done when the author was visiting AT&T Labs–Research, Florham Park, NJ 07932, USA.

‡ This work was done when the author was visiting AT&T Labs–Research, Florham Park, NJ 07932, USA.

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 [18] 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 [27, 26, 28, 16]. 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., [1, 11]). We demonstrate, using a real application 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 providing read/write interactive access to LDAP directories, *the LDAP query language is woefully inadequate for the new generation of DEN applications.* For example, one cannot identify the highest priority policy in the directory that matches a given profile, using an LDAP query. 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 4–6, 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 7. Our central results are that: (a) the query languages exhibit a strict hierarchy of expressive power; and (b) queries written in any of the languages can be evaluated with time and I/O complexities that are linear in the size of the inputs to the query.

After a survey of related work in Section 8, we conclude with a discussion in Section 9.

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., [1, 11]). In this section, we introduce and motivate one running example, to be used throughout the paper, from actual DEN applications that we have studied.

Example 2.1 [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, car phone, etc. The Telephony Over Packet networkS (TOPS) project [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. 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.[1]

Each TOPS 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.

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 QHPs 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 Directory Data Model

In this section, we present a formal description of the core of a scalable network directory data model, based

[1] More generally, a policy in a directory enabled network application defines the desired behavior between multiple objects, and has two 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.

on LDAP [27] and DNS [22], that is particularly suitable for DEN applications. We then revisit our motivating application, and illustrate the modeling of its 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`.[2] 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, values of this 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.[3] ∎

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 share the same type.

3.2 Directory Instance

Just as the relational model uses relations as a single uniform data structure, 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:

1. $R \subset \mathcal{R}$ is a finite set of directory entries,
2. the function $class : R \rightarrow 2^C$ associates with each directory entry a non-empty set of classes from C, to which it belongs,
3. $val : R \rightarrow 2^{A \times dom(\mathcal{T})}$ is a function that associates with each directory entry a set of (attribute, value) pairs[4], s.t. the following conditions are satisfied:
   (a) For each entry $r \in R$, if $val(r)$ contains a pair (a, v) then 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 only if the attribute a is an allowed attribute for at least one of r's classes, and the value v is of the right type.
   (b) For every class name $c \in C$, for every directory entry $r \in R$, $($`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.
4. $dn : R \rightarrow$ `distinguishedName` 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 distinguished names and fully qualified file names in the UNIX system. While the latter 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 `distinguishedName` $\in \mathcal{T}$, entries can have attributes whose value is the dn of some other entry and, hence, can serve as directory entry references.

Directory entries are the basic units for holding information in the directory data model, similar to

[2] 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.

[3] An LDAP directory schema distinguishes between *required* and *allowed* attributes. This distinction is not relevant to the contributions of this paper.

[4] 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.

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 entries 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).[5] 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 [22].

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[6] 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 Application Revisited

For our motivating network application, the higher levels of the directory information forest correspond to the DNS domain and host name hierarchy. Details are omitted for lack of space.

Example 3.1 [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 [3]. When the TOPS data is modeled in the 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`.[7]

The root of such a subtree is a directory entry with the profile of the TOPS subscriber, 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: 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. Sample call appearance entries corresponding to Jagadish's working hours QHP are: his office phone number, which has a higher priority; and his secretary's office phone number, which has a lower priority. On the other hand, his voice messaging mailbox may be the only call appearance specified corresponding to his weekend QHP.

Note that 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 Advantages of Network Directories

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

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, the directory data model can represent and manipulate the heterogeneity inherent in real-world networked entities in a very easy, natural and flexible manner, which is critical for ensuring the autonomy of the different directory servers: (a) An entry can specify values for attributes in the definitions of *any* of its classes, without requiring a single (most-specific) class to contain this union of attributes in its definition. (b) Different entries belonging to the same set of classes may contain very different attributes. (c) A directory entry can have multiple values for an attribute. In

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

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

[7] 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.

comparison, the relational and object-oriented models are considerably more rigid and too homogeneous for networked applications.

The directory data model shares the flexibility of the recently proposed models for semi-structured data (see, e.g., [2, 8, 14]), 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 be described easily by having attributes "pointing" to the referenced entries.

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 [27, 16, 17]. 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., `priority` $<$ `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 r satisfies an atomic filter F, denoted $r \models F$, if *at least one* of the (attribute, value) pairs of r satisfies F. For example,

$$r \models (a = *) \iff \exists(v)((a, v) \in val(r))$$

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*, as described below:

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

We require that atomic queries can be evaluated efficiently. This is not a restrictive assumption, since the atomic queries considered above are all supported by LDAP, and can be evaluated with the help of B-tree indexes for integer and distinguishedName filters, and trie and suffix tree indexes [21] 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 a parenthesized prefix notation. 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 [Use of Different Base Entries] 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 7); 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 [19] describe an elegant table-driven algorithm, with linear I/O complexity, that takes sorted input lists and computes a sorted output list for this task.

For reasons that will become obvious when we consider the evaluation of 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* [28].

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 some 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=orgUnit)`". 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=orgUnit)
   (dc=att, dc=com ? sub ? surName=jagadish))
```

This query returns each entry that satisfies the first operand of the binary `children` operator, and has at least one child entry that satisfies the second operand of the binary `children` operator.

Similar examples also arise when trying to model and unambiguously locate organizational and personal lists in directories; see [20] for more details. ∎

Example 5.2 [Selecting Path Constrained Ancestors and Descendants] The hierarchical organization of directory entries allows `dcObject` entries to be children of other `dcObject` entries, and `orgUnit` entries to be children of other `orgUnit` entries. One often wishes to locate the *closest* `orgUnit` ancestor entry of another directory entry.[8] Consider, for example, the query "Which organizational units in AT&T contain entries with `surName=jagadish`"? The above query can, for example, be specified as:

```
(d_c(dc=att, dc=com ? sub ? objectClass=orgUnit)
   (dc=att, dc=com ? sub ? surName=jagadish)
   (dc=att, dc=com ? sub ? objectClass=orgUnit))
```

This query returns each entry r_1 that satisfies the first operand of the ternary $\texttt{descendants}_c$ operator, and has at least one descendant entry r_2 that satisfies the second operand of $\texttt{descendants}_c$, 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 $\texttt{descendants}_c$. ∎

[8] Using the `descendants` operator, one could locate *all* the `orgUnit` ancestor entries.

5.2 Hierarchical Selection Formalized

$\mathcal{L}_1$ extends $\mathcal{L}_0$ with six operators, c, p, d, a, d_c and a_c, whose semantics are described below.

Definition 5.1 [Semantics of Hierarchical Selection Queries] The semantics of the six hierarchical selection operators is given in Figure 1. ∎

5.3 Evaluating Hierarchical Selection Operators

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

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 [20]. 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 [20], and we paraphrase their argument below because it will help us in understanding the correctness of subsequent algorithms in this 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 $\texttt{descendants}_c$.

$$\mathcal{M}(c\ Q_1 Q_2) \stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge\ r_2 \text{ is a child of } r_1))\}$$
$$\mathcal{M}(p\ Q_1 Q_2) \stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge\ r_2 \text{ is a parent of } r_1))\}$$

$$\mathcal{M}(d\ Q_1 Q_2) \stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge\ r_2 \text{ is a descendant of } r_1))\}$$
$$\mathcal{M}(a\ Q_1 Q_2) \stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge\ r_2 \text{ is an ancestor of } r_1))\}$$

$$\mathcal{M}(d_c\ Q_1 Q_2 Q_3) \stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge r_2 \text{ is a descendant of } r_1 \wedge (\nexists(r_3)(r_3 \in \mathcal{M}(Q_3) \wedge r_3 \text{ is a descendant of } r_1 \wedge\ r_2 \text{ is a descendant of } r_3))))\}$$

$$\mathcal{M}(a_c\ Q_1 Q_2 Q_3) \stackrel{\text{def}}{=} \{r_1 \mid r_1 \in \mathcal{M}(Q_1) \wedge (\exists(r_2)(r_2 \in \mathcal{M}(Q_2) \wedge r_2 \text{ is an ancestor of } r_1 \wedge (\nexists(r_3)(r_3 \in \mathcal{M}(Q_3) \wedge r_3 \text{ is an ancestor of } r_1 \wedge\ r_2 \text{ is an ancestor of } r_3))))\}$$

Figure 1: Semantics of Hierarchical Selection Queries

```
Algorithm ComputeHSPC (op, L1, L2) {
   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 */

   /* Phase 1: each entry in L1 is associated with the number of its parents and children in L2 */
   Initially stack S is empty, entry rl = firstElement(L1, L2), and label(rl) = {i | rl ∈ Li}.
      /* rl points to the first entry in the lexicographic merge of L1 and L2 */
   repeat
      below(rl) = 0, above(rl) = 0.
      if (stack S is empty) push rl on top of stack S, and rl = nextElement(L1, L2).
      else
         let rt be the entry at the top of the stack S.
         if (rt is an ancestor of rl)
            if ((2 ∈ label(rl)) and (rt is a parent of rl)) above(rt) = above(rt)+1.
            if ((2 ∈ label(rt)) and (rt is a parent of rl)) below(rl) = 1.
            push rl on top of stack S, and rl = nextElement(L1, L2).
         else
            if (1 ∈ label(rt)) associate values (above(rt), below(rt)) with entry rt in list L1.
            pop stack.
   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 rl = firstElement(L1).
   repeat
      if ((op is p) and (below(rl) > 0)) output rl.
      else if ((op is c) and (above(rl) > 0)) output rl.
      rl = nextElement(L1).
   until (all entries in L1 have been processed).
}
```

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

$$ws_Q(r_1) \stackrel{\text{def}}{=} \{r_2 \mid r_2 \in \mathcal{M}(Q_2) \wedge\ r_2 \text{ is a parent of } r_1\} \text{ if } op \text{ is } p$$

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

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

Figure 3: Semantics of Aggregate Selection Queries

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, ComputeHSAD, for computing `ancestors` and `descendants`. Details are omitted for lack of space.

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, ComputeHSAD$_c$, for computing $\texttt{ancestors}_c$ and $\texttt{descendants}_c$.

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 algorithm incurs 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 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 ComputeHSAD$_c$ 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 query handling profile for a given TOPS subscriber, 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 some examples next.

6.1 Illustrative Examples

Example 6.1 [Simple Aggregate Selection] The query "Find the query handling profiles specified for the TOPS subscriber `uid=jag, ou=userProfiles, dc=research, dc=att, dc=com` that are applicable for multiple days of the week" can be expressed as follows:

```
(g (uid=jag, ou=userProfiles, dc=research,
     dc=att, dc=com ? sub ? objectClass=QHP)
   count(daysOfWeek) > 1)
```

The aggregate selection filter "`count(daysOfWeek) > 1`" is applied to each of the directory entries that satisfy "(uid=jag, ou=userProfiles, dc=research, dc=att, dc=com ? sub ? objectClass=QHP)". The aggregate term `count(daysOfWeek)` is used to associate each entry with a single value that is the number of values of the `daysOfWeek` 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 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

$\mathcal{L}_2$ extends $\mathcal{L}_1$ with one new operator g, and allows each of the six hierarchical selection operators in $\mathcal{L}_1$ to take an aggregate selection filter as a final operand. 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(priority)`, or (c) an *entry-set aggregate* of the forms `min(min(priority))` 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:

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

Given sets of directory entries $R_1, R_2 \subseteq R$, a function $f : R_1 \rightarrow 2^{R_2}$ 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 $\texttt{count}(\$1)$, the result of applying esa on the (R_1, R_2, f) triple, denoted by $esa[R_1, R_2, f]$, is given by:

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

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*, in Figure 3. The other cases are similar.

Finally, we define the *semantics of structural aggregate selection queries* matching "$(op\ Q_1\ Q_2\ aa_1\ \theta\ aa_2)$" and "$(op\ Q_1\ Q_2\ Q_3\ aa_1\ \theta\ aa_2)$" in Figure 3, 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(priority)` 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(priority))` can also be incrementally computed, using techniques similar to those described in Ross et al. [24], 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 with I/O complexity $O(\frac{|L_1|}{B})$, where B is the blocking factor.* ▮

6.4 Evaluating Structural Aggregate Selection

Each of the 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. 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 [24] can be computed in this fashion.

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 Comparative Assessment

7.1 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 7.1 $\mathcal{LDAP} \subset \mathcal{L}_0 \subset \mathcal{L}_1 \subset \mathcal{L}_2$. ▮

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

Theorem 7.2 *(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. The second is efficiency of evaluation: to see this, note that (p Q_1 Q_2) can be expressed as follows:

(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.

7.2 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 theorem.

Theorem 7.3 *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 indexes. 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 $O(\frac{|L|}{B})$.

7.3 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.

8 Related Work

The network directory model is a hierarchical information model, which should remind readers of the early hierarchical data model (see, e.g., [25]) 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. [19] 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. [20] 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 lists in an LDAP directory. We build upon the works of [19] and [20], 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 [12], Hy+ [13], and WebSQL [4]), and the recently proposed models for semi-structured data (see, e.g., Lorel [2], UnQL [8] and StruQL [14]), while at the same time effectively addressing the specific needs of network directory applications. A significant difference is that graph-based and semi-structured models do not typically give a first class status to the key features of our model, such as hierarchical namespaces, and the definition of node contents as sets of (attribute, value) pairs. While these can be expressed within the general framework of graph-based and semi-structured models using graphs with labeled nodes and/or edges, making them first class components of the model enables better analysis of instances and optimization of queries in the specific context of network directories. Similarly, while it is true that the query languages for graph-based and semi-structured data can express many of the operators of our languages (and much more), focusing on the specific operators that are important to the context of network directories allowed us to design very efficient evaluation algorithms for these operators.

Several researchers have recently proposed schemas for semi-structured data, e.g., graph schemas [7], data guides [15], unary datalog schemas [23], 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.

9 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] Directory enabled networks ad hoc working group. http://www.murchiso.com/den/.

[2] 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.

[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 on Selected Areas in Communications*, 17(1):91–108, 1999.

[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.

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

[12] 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.

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

[14] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, Sept. 1997.

[15] 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.

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

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

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

[19] 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.

[20] 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.

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

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

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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.