Using LDAP Directory Caches

Sophie Cluet*
INRIA Rocquencourt
Sophie.Cluet@inria.fr

Olga Kapitskaia
AT&T Labs–Research
olga@research.att.com

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

Abstract

LDAP (Lightweight Directory Access Protocol) directories have recently proliferated with the growth of the Internet, and are being used in a wide variety of network-based applications to store data such as personal profiles, address books, and network and service policies. These systems provide a means for managing heterogeneity in a way far superior to what conventional relational or object-oriented databases can offer. To achieve fast performance for declarative query answering, it is desirable to use client caching based on semantic information (instead of individual directory entries).

We formally consider the problem of reusing cached LDAP directory entries for answering declarative LDAP queries. A semantic LDAP directory cache contains directory entries, which are semantically described by a set of query templates. We show that, for conjunctive queries and LDAP directory caches with positive templates, the complexity of cache-answerability is NP-complete in the size of the query. For this case, we design a sound and complete algorithm for cache-answerability based on a suite of query transformations that capture the semantics of LDAP queries. We demonstrate the practicality of this algorithm for real applications with a performance evaluation, based on sample queries from a directory enabled application at AT&T Labs. When the query templates in the cache contain negation, we show that the complexity of cache-answerability of conjunctive LDAP queries is co-NP complete in the size of the schema and query templates in the semantic description of the cache. Finally, we identify natural restrictions on the nature of the semantic descriptions for polynomial-time cache-answerability.

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

1 Introduction

LDAP (Lightweight Directory Access Protocol) network directories have recently proliferated with the growth of the Internet, and a large number of directory server implementations are now available from companies such as Critical Angle, Lucent, Microsoft, Netscape, Novell, Sun and Tandem (see [10] for a survey). They are being used in a wide variety of network-based applications to store not only address books and contact information for people, but also personal profiles, network resource information, and network and service policies. 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 by distinguished name or using declarative (database-style) queries. These systems provide a means for managing 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.

To achieve fast performance and high availability in such systems, it is desirable to cache information close to the applications that access the directory information. Such client caching has been established as an effective way to scale the performance of a client-server database architecture (see, e.g., [6]). Physical caching of directory entries occurs in LDAP directories, but the reuse of cached data is currently restricted to access by distinguished name (similar to object references in object-oriented databases), not to access by declarative queries.

There has been much recent interest in the area of client caching to efficiently handle declarative queries using semantic information. The reasons for using the semantic information include better use of the client cache, more autonomy of the client, and reduced client-server communication [14, 12, 3, 4].

1.1 Contributions

In this paper, we formally consider the problem of reusing cached LDAP directory entries for answering LDAP queries, and make the following contributions:

- We present the notion of a *semantic LDAP directory cache*, which contains a set of (projections of) directory entries, along with a semantic description, consisting of the directory schema and a set of query "templates", of the cached data (Section 3).
- We develop a suite of query transformations that capture the semantics of LDAP queries, and design a sound and complete algorithm for determining whether a conjunctive LDAP query is *cache-answerable* using positive query templates (Section 4).
- We show that, for conjunctive queries and LDAP directory caches with positive templates, the complexity of cache-answerability is NP-complete in the size of the query (Section 5). We identify natural restrictions on the nature of the semantic descriptions for polynomial-time cache-answerability. We demonstrate the practicality of our algorithm for real applications with a preliminary performance evaluation, based on sample queries from a directory enabled application at AT&T Labs.
- When the query templates in the cache contain negation, we show that the complexity of cache-answerability of conjunctive LDAP queries is co-NP complete in the size of the schema and query templates in the semantic description of the cache (Section 6).

1.2 Related Work

The various approaches for utilizing semantic information about the data hidden in queries in the caching client can be broadly categorized into three groups: (a) View caching architectures, such as ADMS± [14] and Predicate Caching [12], that "hide" the overlap of query results. (b) "Chunking" architectures, where the database is partitioned into groups of tuples, which are clustered together [4]. (c) Semantic caching architectures, where the cached data is dynamically divided into similar chunks, based on the actual queries that bring in the data [3]. What these approaches have in common, is that they store data brought in by queries, using information from the queries to manage the cache. Furthermore, they all avoid replicating data at the client. Our caching architecture decouples the semantic cache description from the cached directory data, and is closer in spirit to the first group of architectures.

Cache-answerability is related to the problem of finding complete rewritings of a query using views (see, e.g., [13, 15]). A key conceptual difference arises due to the fact that we do not seek a rewriting of the original query in terms of the (views in the) semantic cache description, but want to evaluate the original query against the cached directory instance. Further, techniques for solving the problem of finding complete rewritings in relational databases do not apply here because of the significant differences wrt the data model and query language.

2 The LDAP Directory Data Model and Query Language

We formally present here the core of the LDAP directory data model and query language [16, 8, 9], building on the approach of Jagadish et al. [11].

2.1 LDAP 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 type `string`. We also assume the existence of a complex type `distinguishedName` $\in \mathcal{T}$, whose domain consists of sequences of sets of pairs in $\mathcal{A} \times dom(\mathcal{T})$.

Definition 2.1 [LDAP Directory Schema] A *directory schema* is a 5-tuple $S = (C, A, \tau, \rho, \alpha)$ 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) $\tau : A \to \mathcal{T}$ is a function that associates a type with each attribute, such that $\tau($`objectClass`$) =$ `string`; and (d) $\rho : C \to 2^A$ and $\alpha : C \to 2^A$ are functions that associate sets of attributes with each class name, such that $\forall c \in C : \rho(c) \cap \alpha(c) = \emptyset$.

Given a directory schema $S = (C, A, \tau, \rho, \alpha)$, we denote by $\lambda(S)$, the *relaxed schema* $(C, A, \tau, \rho', \alpha')$, where (i) $\forall c \in C : \rho'(c) = \emptyset$, and (ii) $\forall c \in C : \alpha'(c) = \rho(c) \cup \alpha(c)$. ∎

For a class $c \in C$, we call $\rho(c)$ (resp., $\alpha(c)$) the set of *required* (resp., *allowed*) attributes of c. Given a directory schema $S = (C, A, \tau, \rho, \alpha)$, and an attribute $a \in A$, we denote the set of classes that have the attribute a by $\gamma(a)$, and use $\gamma_r(a)$ to denote the set of classes in which a is a required attribute.

We use the schema in Figure 1, a fragment of a real-life LDAP directory schema used at AT&T Labs, for all the examples in the paper.

2.2 LDAP Directory Instance

Just as the relational model uses relations as a single uniform data structure to represent information, the LDAP model uses a *forest* as a single data structure.

```
class external          class inetOrgPerson   class lip             class residentialPerson
    requires                requires              requires              requires
        objectClass,            objectClass,          objectClass,          objectClass,
        mail,                   sn,                   uniqueId,             sn,
        uniqueId,               cn,                   cn                    cn
                                                                            locality
    allows                  allows                allows                allows
        cn                      tel,                  mail,                 tel
                                locality,             sn,
                                mail                  gender,
                                                      locality,
                                                      tel
```

Figure 1: Schema of Example LDAP directory

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 2.2 [LDAP Directory Instance] A *directory instance* of a directory schema $S = (C, A, \tau, \rho, \alpha)$ 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[1], s.t. the following conditions are satisfied:

1. For each entry $r \in R$, if $val(r)$ contains a pair (a, v) then the attribute a is an attribute for at least one of r's classes, and the value v is of the right type.

2. Each entry $r \in R$ contains at least one value for each of the required attributes of its classes.

3. For each entry $r \in R$, the classes that r belongs to must be the values of r's `objectClass` attribute.[2]

(d) $dn : R \rightarrow$ `distinguishedName` is a function that associates with each directory entry r a sequence of sets of (attribute, value) pairs, referred to as the *distinguished name* of r. This is used as a key to identify directory entries.[3] ∎

[1] 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.

[2] Thus, every directory entry must have values for the `objectClass` attribute, even though the schema does not mandate `objectClass` to be a required attribute of any class.

[3] See Appendix A to understand why this key is a sequence of sets of (attribute, value) pairs, instead of just being a set of (attribute, value) pairs as in conventional databases.

Distinguished names are used to induce a hierarchy among entries in the directory instance, as explained in Appendix A. Note that this is quite different from an object-oriented class hierarchy, which is a hierarchy among schema elements. We do not go into details here since they are not germane to our results and techniques.

The (attribute, value) pairs of a sample directory entry belonging to multiple classes, for the schema of Figure 1, are: {(`objectClass`, `inetOrgPerson`), (`objectClass`, `external`), (`sn`, `cluet`), (`sn`, `verduron`), (`mail`, `sophie.cluet@inria.fr`), (`cn`, `sophie cluet`), (`uniqueId`, `scluet`), (`tel`, `33.1.39.63.58.20`)}.

2.3 The LDAP Query Language

LDAP directory entries can be accessed using database-style search functions as described below.

Definition 2.3 [Filter] An *atomic filter* for a directory schema $S = (C, A, \tau, \rho, \alpha)$ is of the form $\phi(a)$, where $a \in A$ is an attribute, and ϕ is a predicate of $\tau(a)$. A *boolean filter* for a directory schema $S = (C, A, \tau, \rho, \alpha)$ is a boolean combination of atomic filters, using the standard boolean operators: `and` (&), `or` (|), `not` (!). ∎

Conjunctive filters and positive conjunctive filters are defined in the usual manner. For notational simplicity, we use *filter* to refer to either an atomic filter or a boolean filter. For concreteness, we use *exact match*, *presence*, and *substring match* atomic filters for the base type `string` in our examples, and we use a parenthesis-prefix notation for boolean filters. Thus, "`(sn=cluet)`", "`(mail=*)`" and "`(tel=973360*)`" are all atomic filters,[4] and the filter "`(& (sn=cluet) (! (tel=973360*)))`" is a boolean filter.

[4] Note that the presence atomic filter "`(mail=*)`" essentially captures the condition that the directory entry has *at least one* value for the attribute `mail`.

Definition 2.4 [Free-Query] An *LDAP free-query* for a directory schema $S = (C, A, \tau, \rho, \alpha)$ is a 2-tuple $q = (f_q, A_q)$, such that:

- f_q is a filter, referred to as the *query filter*, and
- $A_q \subseteq A$ are the *projected attributes* of the query, such that `objectClass` $\in A_q$. ∎

The answer to a free-query is a set of zero or more responses, each consisting of an entry that "satisfies" the query filter, restricted to the projected attributes (complete with all associated values) of the query; this is analogous to a select-project query, from the relational database point of view. We formalize this intuition below.

Definition 2.5 [Query Answer] Given a directory schema $S = (C, A, \tau, \rho, \alpha)$, the answer to a free-query $q = (f_q, A_q)$ against a directory instance $I = (R, class, val, dn)$, denoted $q(I)$, is a directory instance $(R_q, class, val_q, dn)$ of the relaxed schema $\lambda(S)$, where:

- $R_q \subseteq R$ is the set of all directory entries that satisfy filter f_q, where:
  1. a directory entry *satisfies an atomic filter* if *at least one* of the (attribute, value) pairs of the entry satisfies the filter.
  2. satisfaction of a boolean filter by a directory entry has the obvious set-theoretic meaning.
- $\forall r \in R_q : val_q(r) = \{(a, v) \mid (a, v) \in val(r) \wedge a \in A_q\}$. That is, $val_q(r)$ is the projection of $val(r)$ to the projected attributes of the query. ∎

It is important to note that the projected attributes of a query must include the attribute `objectClass` to ensure that the answer to a query is itself a directory instance, i.e., the *closure* property is satisfied. When $A_q = A$, we say that q is *projection-free*.

LDAP queries are, in general, *scoped* by restricting the applicability of the free-query to a portion of the forest; some details are presented in Appendix B Because of space limitations, we consider only free-queries in this paper. The extensions of our algorithms and results to scoped-queries are not too difficult.

In the rest of this paper, we use *LDAP query* and *query* to refer exclusively to LDAP free-queries.

2.4 Advantages of LDAP Directories

There are two main advantages of LDAP over relational and object-oriented models.

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

The LDAP directory data model shares the flexibility of the recently proposed models for semi-structured data (see, e.g., [1, 2, 5]), while at the same time effectively addressing the specific needs of network directory applications.

3 LDAP Directory Caches

A client cache stores a subset of the data available at the server, along with meta-data that succinctly identifies this subset. For example, an object cache stores objects in the cache and maintains a list of cached object identifiers [6].

3.1 Directory Cache

The LDAP directory cache stores a subset of the data available at the directory server, along with a semantic description (server directory schema, and query "templates") of this data. We formalize this intuition below.

Definition 3.1 [Directory Cache] We define a *directory cache* of an instance $I = (R, class, val, dn)$ of a directory schema $S = (C, A, \tau, \rho, \alpha)$ as a 3-tuple $DC_I = (S, QT, I_c)$, where:

- $QT = \{qt, \ldots\}$ is a set of LDAP queries, each of which is referred to as a *template*.
- $I_c = (R', class, val', dn)$ is the *cached directory instance* of the relaxed schema $\lambda(S)$ where:
  1. $R' = \bigcup_{qt \in QT} R_{qt}$. That is, the cache contains all and only answers to the query templates.
  2. $\forall r \in R' : val'(r) = \bigcup_{qt \in QT} val_{qt}(r)$. That is, the answers to individual templates are "merged" in the cached directory instance. This is made possible by the fact that an answer to a query has the same distinguished name as in the input directory instance.

We refer to (S, QT) as the *semantic description* of the directory cache. ∎

Such a "merged" directory cache avoids replicating data, and also allows the cached entries to be directly managed as an independent directory.

A template in a cache may be a previously posed query, a generalization of previously posed queries, etc. The exact mechanisms by which templates are created are outside the scope of the paper.

3.2 Answering Queries Using the Cache

Any query q can be evaluated against the cached directory instance I_c. However, the result, $q(I_c)$, may neither be *complete* (because of missing data in the cache), nor be *sound* (because of negation in the query) compared to $q(I)$. In this paper, we address the problem of determining when a query can be soundly and completely answered from the directory cache, using *only the semantic description* (S, QT) associated with the directory cache. We formalize this intuition below.

Definition 3.2 [Cache-Answerability] Given a directory schema S, and a directory cache $DC_I = (S, QT, I_c)$ of a directory instance I, we say that query q is (I, I_c)*-cache-answerable* iff $q(I) = q(I_c)$.

Given a schema S, and a semantic description (S, QT) of the cache, we say that query q is *cache-answerable* iff $\forall$ instances I of schema S, $\forall$ sets of templates $QT' \supseteq QT$, and $\forall$ directory caches (S, QT', I_c) of I, q is (I, I_c)-cache-answerable. ∎

Note that the definition allows for sets of templates $QT' \supseteq QT$, since we would like cache-answerability to be preserved when new templates and new data are added to the cache.

We address the following problem in the paper:

> Given a conjunctive query q, and a semantic description (S, QT) of the cache, where each template in QT is conjunctive, is q cache-answerable?

4 Positive Templates: Algorithms

In this section, we consider the problem of cache-answerability when each template in the semantic description (S, QT) of the cache is a positive conjunctive query, and defer the extension to the case of templates with negation until Section 6.

We build an algorithm to check cache-answerability in four steps. Starting from a simple case of projection-free templates with restricted kinds of directory schemas, we add projection to templates to illustrate the impact of merging entries in the cache. We then consider cache-answerability of queries for full-fledged schemas. Finally, we present an incremental algorithm for cache-answerability for positive templates.

4.1 Projection-Free Templates, Restricted Schemas

Suppose that the schema S contains one single class in which all attributes are allowed, and each template in QT is a positive, projection-free query. Obviously a sufficient condition for query $q = (f_q, A_q)$ to be cache-answerable is that there exists one template $qt = (f_{qt}, A)$, such that f_q is syntactically more restrictive than f_{qt} (denoted $f_q \succ_s f_{qt}$); this can easily be checked after having simplified the filters in a standard way. Such a condition is, however, not necessary since an LDAP query is not syntactically constrained (unlike relational and object-oriented queries) to select entries that belong to some specific class, and need not explicitly mention any class in the schema.

Consider, for example, query q_1 and template qt_1 below, where the schema contains only class `lip` (from Figure 1) with all its attributes as allowed.

q_1 : `((gender=m), {tel, objectClass})`
qt_1 : `((objectClass=lip), {ALL})`

In this case, clearly *all* the directory data is in the cache, and every query (in particular, query q_1) can be cache-answered using template qt_1. Making the schema class explicit in the query, however, gives us a sound and complete algorithm for cache-answerability in our simple case.

Proposition 4.1 *Consider a conjunctive query* $q = (f_q, A_q)$ *and semantic description* (S, QT)*, where* S *has a single class* c_1 *in which all attributes are allowed, and each template in* QT *is a positive, projection-free query. Then, a conjunctive query* $q = (f_q, A_q)$ *is cache-answerable iff* $\exists (qt \in QT) : (\& f_q(\texttt{objectClass} = c_1)) \succ_s f_{qt}$. ∎

4.2 Adding Projection to Templates

We now remove the restriction that templates in QT must be projection-free. Thus, we need to check that: (a) as in the simple case, we have cached all entries that satisfy the query filter, (b) each such cached entry has all the projected attributes of the query. Note that, since entries are merged in the cache, attributes of a cached entry may be obtained from the results of different templates. For example, consider query q_{21} and templates qt_{21} and qt_{22} below:

q_{21} : `((& (gender=m) (cn=jag*)),`
`{cn, objectClass})`
qt_{21} : `((gender=m), {gender, objectClass})`
qt_{22} : `((cn=*), {cn, objectClass})`

Clearly, any entry that is in the result of query q_{21} must satisfy the filters of templates qt_{21} and qt_{22}. The merging of cached entries ensures that any entry that satisfies the filters of both templates would have both the attributes `cn` and `gender` in the cache. Consequently, query q_{21} can be cache-answered using templates qt_{21} and qt_{22}.

Furthermore, in order to check the query filter on each cached entry, (c) each entry must either have all the attributes mentioned in the query filter, or the absence of one or more attributes should not modify the query result. For example, query q_{22} can be cache-answered using the above templates qt_{21} and qt_{22}, since `(! (gender=m))` is satisfied by all cached entries without the `gender` attribute, and we know (because of qt_{21}) that these entries do not have the value `m` for `gender` in the directory.

q_{22} : `((& (cn=sophie*) (! (gender=m))),` `{cn, objectClass})`

Algorithm `SC` (for *syntactic check*) below essentially checks the three conditions (a), (b) and (c) by comparing the query with the templates in the cache. The algorithm makes use of two additional algorithms, `AS` and `FS`, presented in Appendix D. `AS` removes those attributes from the projected attributes of q_d that are guaranteed not to be present in any entry that is an answer to q_d. `FS` simplifies filters in a standard way.

```
Algorithm SC(S, QT, q) {
  /* conjunctive q = (f_q, A_q) */
  q = FS(AS(q))
  if q = ∅ return YES
  A'_q = A_q
  A'_f = attributes in f_q not in A_q
  for each template qt ∈ QT
    if (f_q ≻_s f_qt)
      A'_q = A'_q − A_qt
      A'_f = A'_f − A_qt
      if ((A'_q ∪ A'_f) = ∅) return YES
  if (f_q is positive or A'_q! = ∅) return NO
  let f_q be (&f_pos f_neg)
    /* f_pos is purely positive, f_neg is
    purely negative */
  for each attribute a_i in A'_f
    if (f_pos has conjunct (a_i = v_i))
      return NO
    for each (!(a_i = v_i)) in f_neg
      if (SC(S, QT, ((&f_pos(a_i = v_i)), A_q)) = NO)
        return NO
  return YES
}
```

While the three conditions (a), (b) and (c) above guarantee soundness, they still do not guarantee completeness because of the connection between projection on attribute a and the existence (or non-existence) of attribute a. This is illustrated by query q_{23} below, which can be cache-answered using qt_{21} and qt_{22}, because qt_{22} guarantees that the cache contains all entries that have values for the `cn` attribute.

q_{23} : `((gender=m), {cn, objectClass})`

Algorithm `AC` (for *attribute completion*) below materializes this information in queries. `AC1` splits a query into two conjunctive queries, the answers to one of which will all have no values for this attribute, while the answers to the second query will all have values for this attribute. Note that `AC` generates several conjunctive queries, each of which must be cache-answerable for the original query to be cache-answerable.

```
Algorithm AC(S, QT, q_d) {
  /* conjunctive q_d = (f_{q_d}, A_{q_d}) */
  temp_1 = {q_d}
  for each attribute a_i ∈ A_{q_d}
    temp_2 = ∅
    for each conjunctive query q_d1 ∈ temp_1
      temp_2 = temp_2 ∪ AC1(S, QT, q_d1, a_i)
    temp_1 = temp_2
  return temp_1
}

Algorithm AC1(S, QT, q_d, a_i) {
  /* conjunctive q_d = (f_{q_d}, A_{q_d}) */
  q_1 = (f_{q_d}, A_{q_d} − {a_i})
  q_2 = ((&f_{q_d}(a_i = *)), A_{q_d})
  return {q_1, q_2}
}
```

Theorem 4.1 *Consider a conjunctive query $q = (f_q, A_q)$ and semantic description (S, QT), where S has a single class c_1 in which all attributes are allowed attributes, and each template in QT is a positive query.*

Then, q is cache-answerable iff for each query $q_1 \in$ `AC`$(S, QT, ((\& f_q($`objectClass` $= c_1)), A_q))$, Algorithm `SC`(S, QT, q_1) returns YES. ∎

4.3 General LDAP Schemas

We now consider schemas with several classes, each of which may have both required and allowed attributes. The flexibility of the LDAP model results in query equivalences that go beyond standard logic and that are captured by the following three axioms (see Definition 2.2):

Schema Check : an entry having values for attribute a must belong to at least one class for which a is an attribute.

Required Attributes : an entry must have at least one value for each of the required attributes of its classes.

Class Universality : an entry must belong to at least one class.

We present below two transformation algorithms that together capture the equivalences one can infer from the above axioms.

Algorithm `PE` (for *positive expansion*) below covers the application of Schema Check and Required Attributes. Each positive atomic conjunct of the form $(a = v)$ in the query filter is expanded in two steps: first, we conjoin a class in the schema that has attribute a, and second, required attributes of that class are conjoined to the filter. The utility of Algorithm `PE` is illustrated by query q_3, below, which is cache-answerable using template qt_3, since `gender` is an attribute of only one class (`lip`), for which `uniqueId` is a required attribute.

q_3 : `((gender=m), {cn, objectClass})`
qt_3 : `((uniqueId=*),`
`{cn, gender, uniqueId, objectClass})`

```
Algorithm PE(S, QT, q_d) {
  /* conjunctive q_d = (f_{q_d}, A_{q_d}) */
  temp_1 = {q_d}
  for each positive conjunct (a = v) in f_{q_d}
    temp_2 = ∅
    for each query q'_d in temp_1
      temp_2 = temp_2 ∪ PEO(S, QT, q'_d, (a = v))
    temp_1 = temp_2
  return temp_1
}

Algorithm PEO(S, QT, q_d, (a = v)) {
  /* conjunctive q_d = (f_{q_d}, A_{q_d}) with (a, v)
    in f_{q_d} */
  temp = ∅
  for each class c_j ∈ γ(a)
    q_{d_j} = ((&f_{q_d}(objectClass = c_j)
      PEA(S, QT, c_j)), A_{q_d})
    temp = temp ∪ {q_{d_j}}
  return temp
}

Algorithm PEA(S, QT, c) {
  /* class c */
  temp = TRUE
  for each a_i ∈ ρ(c)
    temp = (&temp (a_i = *))      [(a_i = *) underlined]
  return temp
}
```

Conjuncts that are underlined in a filter need to be treated specially by Algorithm `SC`. Essentially, these conjuncts are used in testing whether the query filter is syntactically more restrictive than a template filter, but the templates *need not* have projected attributes to check these conjuncts; such attributes need to be excluded from the initialization of A'_f in Algorithm `SC`. Checking that a cached entry belongs to a class suffices to guarantee that the original directory entry has values for the required attributes of the class.

Finally, Algorithm `CC` (for *class completion*) covers the application of Class Universality. For example, query q_4 below can be cache-answered using query template qt_4.

q_4 : `((! (uniqueId=*)), {sn, objectClass})`
qt_4 : `((sn=*), {sn, objectClass})`

From the non-existence of attribute `uniqueId`, one can infer that the answers cannot belong either to class `lip` or to `external`, and hence must belong either to class `inetOrgPerson` or to `residentialPerson`, for both of which `sn` is a required attribute.

```
Algorithm CC(S, QT, q_d) {
  /* conjunctive q_d = (f_{q_d}, A_{q_d}) */
  temp_q = ∅
  for each class c_i ∈ C
    q_i = ((&f_{q_d}(objectClass = c_i)), A_{q_d})
    temp_q = temp_q ∪ {q_i}
  return temp_q
}
```

We have presented three transformation algorithms, `PE`, `CC` and `AC`, each of which takes as input a conjunctive query, and (potentially) computes a set of conjunctive queries. Each conjunctive query can then be syntactically checked for cache-answerability using Algorithm `SC`. In the absence of negative conjuncts in templates, no additional transformations need to be applied to determine cache-answerability. The main result of this section is that the three transformations can be combined into a *sound* and *complete* algorithm for cache-answerability using positive templates.

Theorem 4.2 *Consider a conjunctive query q and semantic description (S, QT), where each template in QT is a positive query. Let* `PCA`*(S, QT, q) denote $\bigcup_{q_2 \in \mathtt{CC}(S,QT,q_1)}$* `PE`*$(S, QT, q_2)$, where $q_1 \in$* `AC`*(S, QT, q). Then, q is cache-answerable iff for each query $q_3 \in$* `PCA`*(S, QT, q), Algorithm* `SC`*(S, QT, q_3) returns YES.* ∎

Note that conjoining the filter "$(\mathtt{objectClass} = c_1)$" to the query when testing for cache-answerability in the case of schemas with a single class containing only allowed attributes essentially combined the effects of Algorithms `PE` and `CC`.

4.4 Top-Level Stack-Based Algorithm

The composition of our three transformations may, in general, result in very many conjunctive queries that need to be tested for cache-answerability. We present below a stack-based algorithm, CA (for *cache answerable*), which takes as input a vector that specifies a sequence of transformations to be applied, and alternates the application of the transformations in this sequence with the syntactic check for cache-answerability.

```
Algorithm CA(S, QT, q, OP) {
  /* conjunctive q = (f_q, A_q) */
  /* transformations vector OP = [...] */
  stack = (q, 0)
    /* each stack entry is a pair (query,
    OP index), indicating transformation
    to be applied next to the query */
  while stack is not empty
    (q_d, i) = pop(stack)
    if (SC(S, QT, q_d) = NO)
      if (i ≥ length of OP) return NO
      else
        temp_s = OP[i](S, QT, q_d)
        for each q'_d ∈ temp_s
          push (q'_d, i + 1) onto stack
  return YES
}
```

Theorem 4.3 *Consider a conjunctive query q, transformation vector $OP = [AC, CC, PE]$, and semantic description (S, QT), where each template in QT is positive. Then, q is cache-answerable iff Algorithm* CA*(S, QT, q, OP) returns YES.* ▮

Of course, other sound and complete transformation vectors exist, and determining a transformation sequence that minimizes the expected time/space is outside the scope of the paper.

5 Positive Templates: Efficiency

5.1 Complexity of Cache-Answerability

A consequence of the modeling flexibility of LDAP is that determining whether a positive query is cache-answerable, even using positive, projection-free templates, is hard.

Lemma 5.1 *Consider a conjunctive query q, and a semantic description (S, QT), where each template in QT is a positive query with at most m conjuncts. Then, $\forall(qt \in QT) : f_q \succ_s f_{qt}$ iff $\exists$ m conjuncts in f_q, whose conjunction $f_q^m \succ_s f_{qt}$.* ▮

Theorem 5.1 *Consider a positive query q, and a semantic description (S, QT), where each template is a positive, projection-free query with 2 conjuncts, and each query attribute appears in at least 3 classes. Then, determining if q is cache-answerable is NP-complete in the size of the query.* ▮

The proof first shows that under the conditions described in the theorem, our problem is equivalent to a two-coloring problem on a multi-partite graph. We then show NP-hardness by reduction from the NP-complete problem, EXACT 3-COVER (SP2 in [7]) to an instance of the two-coloring problem on a multi-partite graph. Details are omitted for reasons of space.

5.2 Polynomial-Time Cases

There are some straightforward cases where cache-answerability can be established in polynomial-time, for example, when the schema contains a single class, or when each template filter is atomic. In this section, we describe a less trivial class relying on two categories of restricted atomic filters:[5]

Category 1 : Filters of the form $(a = v), v \neq *$, for attributes a (other than `objectClass`) that are required in some class.

Category 2 : Arbitrary filters, $(a = v)$, for attributes a (other than `objectClass`) that are not required in any of its classes.

Definition 5.1 [Real-Restricted Query] A *real-restricted query* $q = (f_q, A)$ is a positive projection-free query, where each atomic filter in f_q is either a category 1 or a category 2 filter. ▮

Theorem 5.2 *Consider a conjunctive query $q = (f_q$, $A_q)$ and a semantic description (S, QT), where each template in QT is a real-restricted query. Then q is cache-answerable iff $\exists(qt \in QT) : f_q \succ_s f_{qt}$. Further, cache-answerability of q can be determined in polynomial-time.* ▮

5.3 Experimental Results

In order to evaluate the practical benefits of our approach we have implemented the algorithms and conducted many experiments with sample queries from a directory enabled application at AT&T Labs. We observed that these real-world queries have few conjuncts. Our experimental results show that: (a) our algorithms are very fast, even with a large number of templates; and (b) the execution time of our algorithm is small compared to the time for evaluating the query against the directory.

Scalability We have implemented our query transformation algorithms in a scalable fashion. The templates are pre-processed and stored in a hash table based on the set of classes that contain the attributes mentioned in a template. This allows for the fast retrieval of relevant templates during query transformation, and allows

[5] Note that filters of these categories cannot be inferred by our transformation rules.

for good scalability with the increase in the number of the templates. For our experiments, we used an augmented version of the complete Netscape 3.1 Directory Server schema (65 classes, 302 attributes). We varied the number of templates from 5 to 55, each having one to three conjuncts. With 55 templates, the average execution time of multiple runs of 45 queries, each having one to three conjuncts, randomly selected from our application, is 0.018s.

Average Response Time We used an LDAP directory on the local machine as our client cache, and an LDAP directory on a distant machine (but within the same local area network) as our directory server, each with a few thousand entries, and indices on all relevant attributes to efficiently answer queries. The following table gives the minimum, maximum and average of AlgoRatio and CacheRatio for 5, 25 and 55 templates, where AlgoRatio is the ratio of the execution time of our algorithm to the query evaluation time at the directory server, and CacheRatio is the ratio of the total execution time at the cache (algorithm + query evaluation) to the query evaluation time at the directory server. Note that as the number of templates in the cache increases, these ratios monotonically increase reflecting the additional processing that must be performed at the client cache. However, the averages increase slowly reflecting the scalability with the increase in the number of the templates.

	AlgoRatio			**CacheRatio**		
	min	*max*	*avg*	*min*	*max*	*avg*
5	0.0012	0.042	0.012	0.031	0.169	0.094
25	0.0012	0.051	0.015	0.031	0.177	0.098
55	0.0012	0.226	0.044	0.031	0.318	0.126

Thus, the queries that miss the cache would see an average increase of 4.4% in their query response time (due to the additional cost of executing our algorithm), while the queries that hit the cache see an average decrease of 87.4% in their query response time. Thus, assuming a 50% hit rate of the cache (with 55 templates), we achieve 41.5% reduction in the average query response time. With larger directories, the ratios would be even smaller, resulting in still higher benefits.

6 Adding Negation to Templates

6.1 Complexity of Cache-Answerability

When templates have negation in their filters, determining whether a conjunctive query is cache-answerable is much harder than when the templates are positive.

Theorem 6.1 *Consider a conjunctive query q, and a semantic description (S, QT), where each template in QT is conjunctive. Then, determining if q is cache-answerable is co-NP complete in the size of the semantic description.* ▮

We show co-NP hardness by reducing the co-NP complete problem 3-TAUTOLOGY (LO8 in [7]) to our problem. Details are given in Appendix C. Essentially, each source of flexibility of the LDAP data model is *independently* sufficient for establishing the hardness of our problem.

6.2 Projection-Free Templates

We consider here the case where each template is projection-free. When templates contain negation, e.g., if $(!(a = v))$ appears in one template, and $(a = v)$ in another, some simplification may be performed. For example, query q_5 below can be cache-answered using templates qt_{51} and qt_{52}.

q_5 : `((cn=olga*), {cn, objectClass})`
qt_{51} : `((& (cn=olga*) (mail=*)), {ALL})`
qt_{52} : `((& (cn=*) (! (mail=*))), {ALL})`

Performing all possible template simplifications in such situations can result in an exponential blow-up in the number of templates in the cache. Hence, Algorithm `NC`, given in Appendix E, instead modifies queries so that they can be checked for cache-answerability.

When the query contains negation, new equivalences can be inferred from negative filters. (When the templates are positive, we can show that these additional inferences are not needed to determine cache-answerability.) Algorithm `NE` (for *negative expansion*), given in Appendix E, covers the application of the axioms Schema Check and Required Attributes on negative filters. For example, query q_6 below can be cache-answered using template qt_6 since entries without values for the `uniqueId` attribute cannot belong to class `lip` and, accordingly, do not have values for the `gender` attribute.

q_6 : `((& (cn=olga*) (! (uniqueId=*))),`
`{cn, objectClass})`
qt_6 : `((& (cn=*) (! (gender=*))), {ALL})`

Combining these two transformations with the previous transformations (except `AC`, which is not needed for projection-free templates), we can establish the following result.

Theorem 6.2 *Consider a conjunctive query q, transformation vector $OP = [NC, NE, CC, PE]$, and semantic description (S, QT), where each template in QT is a projection-free query. Then, q is cache-answerable iff Algorithm `CA`(S, QT, q, OP) returns YES.* ▮

6.3 Combining Negation with Projection

When templates are arbitrary conjunctive queries, the interaction between templates with negation, and templates with projection can result in our algorithm being unsound for determining cache-answerability. We illustrate this problem with the following example:

q_7 : ((& (sn=*) (! (gender=m))),
{sn, gender, objectClass})
qt_{71} : ((& (sn=*) (! (gender=m))),
{sn, gender, objectClass})
qt_{72} : ((mail=*), {sn, objectClass})

Consider a directory entry with `mail` and `sn` attributes, and value `m` for the `gender` attribute. This entry satisfies qt_{72}, and hence will be in the cache with only the `sn` and `objectClass` attributes. When query q_7 is evaluated against the cached entries, this entry will be part of the answer, which is obviously wrong.

7 Future Work

Many problems remain open, and we only mention a few that we are currently pursuing. How can one determine cache-answerability for arbitrary conjunctive templates? How can one use the cache for answering portions of queries? What criteria does one employ for populating and discarding templates from client caches?

To the best of our knowledge, this paper is the first work on the topic of query caching in LDAP directories, and it opens a useful and challenging area of research for database theoreticians and practitioners.

Acknowledgements

We would like to thank Robert E. Schapire for useful discussions on the complexity of our problem, and the reviewers for their extremely thorough comments which helped improve the presentation of the paper.

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] 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.

[3] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of the International Conference on Very Large Databases*, pages 330–341, Bombay, India, 1996.

[4] P. M. Deshpande, K. Ramaswamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Seattle, WA, 1998.

[5] 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.

[6] M. J. Franklin. *Client data caching: A foundation for high performance object database systems.* Kluwer Academic Publishers, 1996.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

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

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

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

[11] H. V. Jagadish, L. V. S. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Philadelphia, PA, June 1999.

[12] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5(1):35–47, 1996.

[13] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the ACM Symposium on Principles of Database Systems*, San Jose, CA, May 1995.

[14] N. Roussopoulos, C. M. Chen, S. Kelley, A. Delis, and Y. Papakonstantinou. The ADMS project: Views "R" Us. *IEEE Data Engineering Bulletin*, June 1995.

[15] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. In *Proceedings of the International Conference on Very Large Databases*, pages 318–329, Bombay, India, 1996.

[16] 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.

A LDAP Distinguished Names

A *distinguished name* of a directory entry r, denoted $dn(r)$, is a sequence $s_1, \ldots, s_n$ of sets of (attribute, value) pairs. 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 condition: $\forall r, r' \in R : r \neq r' \Rightarrow dn(r) \neq dn(r')$, i.e., $dn(r)$ must be a key.

We use the 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.

Note the resemblance between 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, the LDAP 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` is a defined type, directory entries can have attributes whose value is the *dn* of some other directory entry and, hence, can serve as directory entry references.

B LDAP Scoped-Queries

An LDAP *scoped-query* is, in general, a 4-tuple $q = (b_q, s_q, f_q, A_q)$, where the applicability of the free-query (f_q, A_q) is restricted in two ways:

- First, one can specify a base entry, using the distinguished name b_q, that is the entry relative to which the free-query is to be evaluated.
- Second, one can specify a scope, s_q, which indicates whether the free-query 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`).

C Proof of Theorem 6.1

Given a boolean formula in disjunctive normal form with at most three literals per disjunct, checking if this formula is a tautology is co-NP complete (LO8 in [7]). Call this problem 3-TAUTOLOGY. We reduce 3-TAUTOLOGY to our problem to show co-NP hardness. Consider a boolean formula in disjunctive normal form with three literals per disjunct, with m variables $a_1, a_2, \ldots, a_m$ and n disjuncts. We create a single class c_1 in the schema with $m+1$ allowed attributes $a_0, a_1, \ldots, a_m$; thus, a directory entry may have values for any subset of these $m+1$ attributes. With each disjunct of the boolean formula, say $a_i \wedge \neg a_j \wedge a_k$, we create a conjunctive template of the form $((\&(a_0 = *)(a_i = *)(!(a_j = *))(a_k = *)), \texttt{objectClass})$ in QT; thus, QT has n templates. It is easy to see that the conjunctive query $q = ((a_0 = *), \texttt{objectClass})$ is cache-answerable iff the boolean formula is a tautology. This completes the proof of co-NP hardness.

To show that our problem is in co-NP, we show that the complement of our problem is in NP. It is easy to "guess" a directory entry r that is polynomial in the size of the schema. We can then verify in polynomial time that it is an instance of S, it is an answer to a template as well as an answer to the query.

D Filter and Attribute Simplification

Algorithm `FS` (for *filter simplification*) below eliminates conjunctive queries whose filters are syntactically unsatisfiable, and remove redundancies in syntactically satisfiable filters, in a standard way. Note that all "underlined" filters (which only need to be optionally checked by `SC`) are either of the form $\underline{(a = *)}$ or of the form $\underline{(!(a = *))}$.

`Algorithm FS`(q_d) {
  `/* conjunctive` $q_d = (f_{q_d}, A_{q_d})$ `*/`
  `for each conjunct` $(a = v_1)$ `in` f_{q_d}
    `if (`f_{q_d} `contains` $(!(a = v_2))$`)`
      `if (`v_2 `covers` v_1`) return` $\emptyset$
  `remove all duplicate conjuncts`
  `for each conjunct` $(a = v_1)$ `in` f_{q_d}
    `if (`f_{q_d} `contains` $(a = v_2)$`)`
      `if (`v_2 `covers` v_1`)`
        `remove` $(a = v_2)$ `from` f_{q_d}
  `for each conjunct` $\underline{(a = *)}$ `in` f_{q_d}
    `if (`f_{q_d} `contains` $(a = *)$`)`
      `remove` $(a = *)$ `from` f_{q_d}
  `for each conjunct` $(a = v_1)$ `in` f_{q_d}
    `if (`f_{q_d} `contains` $\underline{(a = *)}$`)`
      `remove` $\underline{(a = *)}$ `from` f_{q_d}
  `for each conjunct` $(!(a = v_1))$ `in` f_{q_d}
    `if (`f_{q_d} `contains` $(!(a = v_2))$`)`
      `if (`v_1 `covers` v_2`)`
        `remove` $(!(a = v_2))$ `from` f_{q_d}
  `for each conjunct` $\underline{(!(a = *))}$ `in` f_{q_d}
    `if (`f_{q_d} `contains` $(!(a = v_2))$`)`
      `remove` $(!(a = v_2))$ `from` f_{q_d}
  `return` q_d
}

Algorithm AS (for *attribute simplification*) removes those attributes from the projected attributes of query q_d that are guaranteed not to be present in any directory entry that is an answer to query q_d.

```
Algorithm AS(q_d) {
  /* conjunctive q_d = (f_{q_d}, A_{q_d}) */
  for each conjunct (!(a = *)) in f_{q_d}
    A_{q_d} = A_{q_d} - {a}
  return {q_d}
}
```

E Negative Completion and Expansion

Algorithm NC (for *negative completion*) below "splits" a query into multiple conjunctive queries, using negative conjuncts in the templates. Applying Algorithm NC to query q_5 using the negative conjunct (! (mail=*)) in template qt_{52} would result in q_5 being split up into two conjunctive queries, each of which can be determined to be cache-answerable by Algorithm SC.

```
Algorithm NC(S, QT, q_d) {
  /* conjunctive q_d = (f_{q_d}, A_{q_d}) */
  temp_1 = {q_d}
  for each (!(a = v)) in f_{qt}, template qt ∈ QT
    temp_2 = ∅
    for each conjunctive query q_{d1} ∈ temp_1
      temp_2 = temp_2 ∪ NC1(S, QT, q_{d1}, (!(a = v)))
    temp_1 = temp_2
  return temp_1
}

Algorithm NC1(S, QT, q_d, (!(a = v))) {
  /* conjunctive q_d = (f_{q_d}, A_{q_d}) */
  q_1 = ((&f_{q_d} (!(a = v))), A_{q_d})
  q_2 = ((&f_{q_d} (a = v)), A_{q_d})
  return {q_1, q_2}
}
```

Algorithm NE (for *negative expansion*) below makes explicit all filters that can be inferred from a conjunction of negative atomic filters.

Applying Algorithm NE to query q_6, followed by filter simplification, would result in query q'_6 below, which is cache-answerable using qt_6.

```
q'_6 : ((& (cn=olga*) (! (uniqueId=*))
    (! (objectClass=lip)) (! (gender=*))
    (! (objectClass=external)))
    {cn, objectClass})
```

```
Algorithm NE(S, QT, q_d) {
  /* conjunctive q_d = (f_{q_d}, A_{q_d}) */
  f_temp = f_{q_d}
  for each conjunct (!(a = *)) in f_{q_d}
    f_temp = (&f_temp NEO(S, QT, a))
  q_temp = ((&f_temp
    NEA(S, QT, f_temp)), A_{q_d})
  return {q_temp}
}

Algorithm NEO(S, QT, a) {
  /* attribute a */
  temp = TRUE
  for each class c_j ∈ γ_r(a)
    temp = (&temp (!(objectClass = c_j)))
  return temp
}

Algorithm NEA(S, QT, f) {
  /* filter f */
  temp_c = ∅
    for each (!(objectClass = c_i)) in f
      temp_c = temp_c ∪ {c_i}
  temp_a = ∅
  for each attribute a_i ∈ A
    if (γ(a_i) ⊆ temp_c)
      temp_a = temp_a ∪ {a_i}
  temp_f = TRUE
  for each attribute a_i ∈ temp_a
    temp_f = (&temp_f (!(a_i = *)))
  return temp_f
}
```