Edge Caching for Directory Based Web Applications: Algorithms and Performance

Apurva Kumar, Rajeev Gupta
IBM India Research Lab
Block 1, IIT Delhi, India
+91-11-26861100
{kapurva,grajeev}@in.ibm.com

ABSTRACT

In this paper, a dynamic content caching framework is proposed for deploying directory based applications at the edge of the network, closer to the client. The framework consists of a Lightweight Directory Access Protocol (LDAP) directory cache and the offloaded application running at a proxy. The LDAP directory cache is an enhanced LDAP proxy server which stores results and semantic information for search requests (*queries*) and answers incoming queries which are semantically contained in them. A simplified query containment approach based on the concept of LDAP *templates* is proposed. Caching algorithms have been proposed which take advantage of referential locality in the access pattern. A generic framework is used to offload the application at the edge and to support prefetching of LDAP queries based on application logic. A real enterprise directory application and real workloads are used to evaluate performance of the caching algorithms. The LDAP directory cache architecture, along with the proposed algorithms can be used to improve performance and scalability of directory based services.

Keywords

LDAP, query caching, templates, query containment, prefetching, application offload, XML.

1. INTRODUCTION

There has been a growth of websites providing dynamic content on the web. Typically dynamic content is generated in response to a user request (*query*) evaluated against a database. Techniques used for traditional (static) content caching are, in general, not applicable for caching dynamic content.

Directories are specialized databases, which are capable of storing heterogeneous real world information in a single instance. The Lightweight Directory Access Protocol (LDAP) provides a means for accessing and managing remote and distributed directories [1,2]. LDAP directories are being used to store address books, contact information, customer profiles, network resource information, policies etc. Directories have assumed special significance in enterprises where a single directory instance containing employee and other organizational records is used by a wide variety of internet and intranet applications. The heterogeneous nature of information stored in directories allows them to be used by a wide variety of applications. However, this also means that an overloaded directory could be a potential bottleneck in an enterprise infrastructure.

Database caching has been established as an effective means to improve performance of client server relational database systems [5-8]. In [5], an architecture for database caching at application servers is presented. In [6,9] an active query-caching framework for form-based proxy caching of database-backed websites is described and significant gains compared to passive query caching are reported.

While the above strongly motivate the need for a framework to be developed for caching/prefetching directory queries for web applications, the mechanisms for database query caching are not directly applicable because of the inherent differences in data models and query languages of directories and relational databases [10,11].

LDAP replication has widely been used for improving performance, scalability and availability of directory based web applications. However, since a typical directory can contain millions of records, the search performance of replicas suffers due to disk access latency [4]. Partial replication is useful only if

applications need to access a part of the directory. An LDAP caching solution, which provides significant query hit ratio, while caching only a small fraction of the records in the directory is thus desirable.

There are two distinct problems associated with LDAP query caching: *(i)* Determining whether an LDAP query is contained in another query (*query containment*), *(ii)* Using *caching algorithms* which make efficient caching, prefetching, cache replacement decisions to maximize the fraction of queries answered from the cache. The complexity of *(i)* is the subject of [12]. The authors consider the general query containment problem for LDAP and show it to be NP-complete in the size of the query. The authors of [13] introduce the notion of generalized queries and propose algorithms for *(ii)*. They use a real directory but synthetic workloads and a single type of query for performance evaluation.

In our work we reduce the complexity of the query containment problem by introducing the concept of LDAP templates (Section 4). For *(ii)* we determine the caching algorithm to be used based on the type of query. The proposed directory cache is an LDAP proxy server extended for query caching. The performance of an LDAP cache implementation has been evaluated for an enterprise directory containing employee records of a large organization and real workloads for an employee white pages application.

Most directory enabled applications are web based and use directories to generate dynamic content in response to HTTP user requests. Resources that are deployed closer to the client are becoming under-utilized with increasing dominance of dynamic content in the web. To reduce client latency and to improve scalability of the origin application server and directory server, there is a need to offload some components of the application to a proxy server at edge of the network where the LDAP cache is located. We use a generic web publishing framework to offload the cacheable component of a directory based application to the edge of the network. The offloaded application can also be used to prefetch LDAP queries based on application logic to improve the cache performance as described in Section 7.2.

The paper is organized as follows: Section 2 describes notations and terminology used in the paper. Section 3 describes the proposed caching framework. Section 4 discusses the query containment problem and proposed solution. Section 5 discusses the caching algorithms. Section 6 describes the set of extensions required in typical directory servers to incorporate query caching. A generic XML based publishing framework and its use in application offload and prefetching is described in Section 7. In Section 8, performance of caching algorithms is evaluated for a real enterprise directory application. Contributions of the work are discussed in Section 9.

2. NOTATIONS

This section introduces briefly some relevant notations and terms used in the rest of the paper. LDAP assumes the existence of one or more directory servers jointly providing access to a *Directory Information Tree* (DIT), which is made of entries. An *entry* is defined as a set of *attribute* value pairs. Each entry has a distinguished name (DN) belonging to a hierarchical namespace. A *directory cache* (or *LDAP cache*) stores a subset of directory entries which correspond to results of cached queries obtained from a *master directory server*.

The functional model adopted by LDAP is one of clients performing protocol operations against servers. LDAP defines three types of operations: query operations, like *search*, *compare*, update operations like *add*, *modify*, *delete* and connect/disconnect operations like *bind*, *unbind*. The most common operation is *search*, which provides a flexible means of accessing information from the directory. The search operation also referred as a *query* consists of the following parameters which represent the semantic information (or *metadata*) associated with a query [1]:

base: A DN that defines the starting point of the search in the DIT.
scope: {BASE, SINGLE LEVEL, SUBTREE}
Specifies how deep within the DIT to search from the *base*.
filter: A boolean combination of *predicates* using the standard operators: AND `(&)`, OR `(|)` and NOT `(!)`, specifying the search criteria.
attributes: Set of required attributes from entries matching the filter.

LDAP filters are represented using the parentheses prefix notation of RFC 2254 [3], e.g.: `(&(sn=Doe)(givenName=John))`. Filters without any NOT operators are called *positive filters*. *Predicates* of the form (*name operator value*) where *operator* $\in \{=, \leq, \geq\}$ are considered. *name* is an attribute name and *value* is its asserted value termed as *assertion value*. Example of predicates are: `(sn=Doe)`, `(age≥30)`, `(sn=smith*)` where "`Doe`"

"30" and "smith*" are assertion values representing equality, range and substring assertions, respectively.

3. LDAP CACHING FRAMEWORK

Figure 1 shows various components of the proposed LDAP caching framework for directory based web applications. The dashed arrows show the path of a client request in the absence of the framework. A remote client request is serviced by a web application which accesses a master directory to answer it.

Using the framework a remote client request is redirected, possibly using DNS redirection, to an edge site closer to the client. The edge site consists of an offloaded application which converts user requests to LDAP queries and sends them to the LDAP cache. Stored metadata for cached queries and the semantic information of an incoming query is used to determine whether it is contained in any of the cached queries. Contained queries are answered locally while other queries are forwarded to the master directory server. This improves performance for remote clients by reducing latency and makes the service more scalable by offloading web and LDAP requests to the edge site.

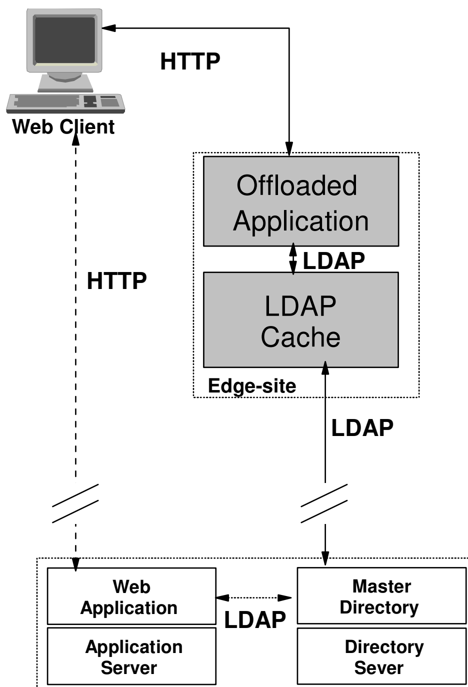

Figure 1: LDAP Caching Framework

4. LDAP QUERY CONTAINMENT

4.1 Query containment problem

An LDAP query Q, is said to be *contained* in another query Q', if all of the following are true:

C1: The *base* of Q is same as or descendant of the *base* of Q'.

C2:

The *scope* of Q' is SUBTREE.

OR

(The *scope* of Q' is same as the *scope* of Q) AND (the *base* of Q' is same as the *base* of Q)

OR

(The *scope* of Q' is SINGLE LEVEL) AND (*scope* of Q is BASE) AND (*base* of Q is a direct descendant of the *base* of Q').

C3: *attributes* in Q is a subset of *attributes* in Q'.

C4: The *filter* of Q is semantically *contained* in the *filter* of Q'.

In general, a contained query is *answerable* (from the cache) if *attributes* of Q in C3 above is interpreted as the union of required *attributes* and the attributes appearing in its search filter. We neglect the impact of directory schema (e.g. mandatory attributes) on the cache answerability problem. Using schema information can provide a more complete cache answerability algorithm. However, as noted in [12] it makes the general problem NP complete in the size of the query.

4.2 General filter containment

An LDAP filter F_1 is *contained* in F_2 if it is not possible for an entry to satisfy F_1 but not F_2. This condition is formalized by Proposition 1.

Proposition 1:

An LDAP query filter F_1 is semantically contained in another query filter F_2 if and only if the expression $F_1 \wedge \neg F_2$ is *inconsistent*. □

For the expression $F_1 \wedge \neg F_2$ to be inconsistent:

$\nexists\, x_1,x_2..x_n$ such that $F_1 \wedge \neg F_2$ is satisfied.

where attribute set $\{x_1,x_2..x_n\}$ is the union of attribute sets appearing in the filters F_1 and F_2.

If $F_1 \wedge \neg F_2 = B_1 \vee B_2 \ldots \vee B_k$ where each B_i is a conjunct of simple predicates, then each B_i should be inconsistent, i.e. the following boolean expression should evaluate to TRUE.

$$(\nexists x_1,x_2..x_n(B_1)) \wedge (\nexists x_1,x_2..x_n(B_2)) \ldots \wedge (\nexists x_1,x_2..x_n(B_k)) \quad (1)$$

where $\nexists x_1,x_2..x_n$ (B_i) represents the condition that B_i is not satisfiable for any values of attributes $x_1,x_2..x_n$ in their valid ranges.

Let the set A_{XY} represent the union of sets of assertion values in LDAP filters X and Y.

Proposition 2:
For positive LDAP filters F_1 and F_2 containing equality and range predicates, the condition for F_1 to be contained in F_2 can be expressed as a boolean expression in conjunctive normal form (CNF) with each simple predicate of the form:
$(a \geqslant b)$ where $a,b \in A_{F1F2}$. □

Sketch of proof: The expression in *(1)* is a conjunct. The condition for each B_i being inconsistent requires that the predicates in B_i should impose an empty range for at least one of the attributes appearing in it. Thus the condition of each B_i being inconsistent is disjunctive and *(1)* can be written in CNF. It is easy to show that a possibly empty range for an attribute x_j imposed by the predicates of B_i is $(a_{xj}, b_{xj}]$, or $[a_{xj}, b_{xj})$ where $a_{xj}, b_{xj} \in A_{F1F2}$. For this range to be empty $a_{xj} \geqslant b_{xj}$.

Example:

Let F_1 be $(a \leqslant p) \wedge (b \geqslant q)$
and F_2 be $(a = x) \vee (b \geqslant y)$
The condition for F_1 to be contained in F_2 is easily seen to be $(q \geqslant y)$, but the example helps in illustrating proposition 2.
Here, $A_{F1F2} = \{p, q, x, y\}$.
F_1 is contained in F_2 if the following expression is inconsistent:

$$F_1 \wedge \neg F_2 = ((a \leqslant p) \wedge (b \geqslant q) \wedge (a > x) \wedge (b < y)) \vee ((a \leqslant p) \wedge (b \geqslant q) \wedge (a < x) \wedge (b < y))$$

$$B_1 = ((a \leqslant p) \wedge (b \geqslant q) \wedge (a > x) \wedge (b < y)).$$

$$B_2 = ((a \leqslant p) \wedge (b \geqslant q) \wedge (a < x) \wedge (b < y)).$$

For B_1 to be inconsistent: $(x \geqslant p) \vee (q \geqslant y)$
For B_2 to be inconsistent: $(q \geqslant y)$
Thus F_1 is contained in F_2 if:

$$((x \geqslant p) \vee (q \geqslant y)) \wedge (q \geqslant y) \Rightarrow (q \geqslant y)$$ □

In the worst case all m predicates in F_1 might have to be compared with all n predicates in F_2. Thus checking containment of an incoming filter with a cached filter requires $O(mn)$ such comparisons.

4.3 Template based filter containment

To reduce the complexity of the problem, we introduce the concept of *LDAP templates*. Most query filters generated by applications are different from other filters of the same type in only their assertion values. A *template* is a filter with one or more assertion values unspecified. A template is also represented using the LDAP filter representation of [3] except that the missing assertion values are replaced by the "_" character. Examples of templates are: `(&(cn=_)(ou=research))`, `(uid=_)`, `(&(sn=_)(givenName=_))`. A substring template for generating queries with the initial string specified is `(sn=_*)`. Templates can be used to generate actual queries by supplying missing assertion values.

In template based query containment, queries belonging to only a specified set of templates, T, are cached and answered. The template based approach has several advantages. Firstly, the number of query comparisons are reduced since queries of a given template A can possibly be answered by only a subset of the templates $T_A \subseteq T$. E.g. a query of template `(&(sn=_)(ou=_))` can not answer a query of template `(sn=_)`. Secondly, conditions for a query a of template A to be contained in a query b of template $B \in T_A$ can be represented as a predetermined generic boolean expression $C_{cnf}(A,B,A_{ab})$ (using a generalization of proposition 2). E.g. query `(age=X)` can be answered by query `(age≥Y)`, if $(X \geqslant Y)$. Thirdly, cache specific parameters used by the caching algorithms can be specified per template. E.g. different cacheability and consistency requirements can be specified for different templates.

The following observation about positive filters belonging to the same template can be made:

Proposition 3:
Let F_1 and F_2 be two positive LDAP query filters belonging to the same template. F_1 is contained in F_2 if each predicate in F_1 is contained in the corresponding predicate of F_2. □

Containment problem for filters of the same template having n predicates using Proposition 3 requires $O(n)$ comparisons of assertion values.

4.4 Query Containment Algorithm

To evaluate whether a query *q*, with filter *f*, is contained in a cache having set of cacheable positive templates *T*, the following steps are performed:

1) Find template of the incoming query filter, say *a*.
   if $a \notin T$, return FALSE.
2) For all templates, $b \in T_a$
   For all queries q_t (with filter f_t) in *b*
   if all of *C1, C2, C3* are satisfied for $Q=q$ and $Q'=q_t$,
   if $a=b$
   if all predicates of *f* are contained in corresponding predicates of f_t, return TRUE.
   else
   if $C_{cnf}(a,b,A_{fft}) = 1$, return TRUE.
   return FALSE.

An important aspect of LDAP query containment is that all comparisons of assertion values should be performed using the correct syntax and matching rules for the corresponding attribute. E.g. assertion values in the two filters `(telephoneNumber=2686-1100)` and `(telephoneNumber=26861100)` are equal according to the equality matching rule described for the commonly used `telephoneNumber` attribute.

The algorithms described in the section can be extended for substring assertions by interpreting substrings as range assertions.

5. LDAP CACHING ALGORITHMS

In this section, we consider improving performance of LDAP caching, for a given cache size, by making efficient caching, prefetching and cache replacement decisions. The directory cache is considered 'hit' by an incoming query if the query is semantically contained by any cached query. The cache can answer such queries without contacting the master server. For caching to be useful, the access pattern should demonstrate locality of reference. In the context of query caching, we define *spatial locality* as accessing of semantically or otherwise related queries in a short time span. Similarly, *temporal locality* is defined as accesses for the same query in a short time span. We extend this to include those accesses, which are contained in previous accesses. In the rest of the section we discuss the proposed caching algorithms.

Query based caching

Incoming user queries are cached if the number of entries in the result set is within specified limits. No prefetching is done. Cache replacement removes the least recently used (LRU) query. □

The limit on number of entries is to restrict the maximum size of a cached query.

The algorithm can not take advantage of spatial locality defined above. By caching user queries themselves, one can not answer queries which are related but not contained in them. Another problem with query based caching is that user queries typically return only a few entries. The number of such queries required to provide a reasonable hit ratio could be very large, thereby increasing query containment costs and storage requirements.

In [13], *super-query caching* (called template caching in their work) is considered and its performance for a real directory but synthetic workloads is evaluated. Generalized form of a user query (super-query) is prefetched when it is estimated to correspond to a hot region. An example of generalized query for the user query `(mail=foo@xyz.com)` is `(mail=*@xyz.com)`. Our version of the super-query caching algorithm which is simpler than [13], is described below:

Super-query caching

A *popularity index* (defined below) is maintained for each of several candidate super-queries. When this index crosses a *threshold*, all the entries corresponding to the super-query are fetched and cached. Cache replacement removes the least popular super-query (and its entries).

The *popularity index* is a measure of hotness of a super-query. It is defined as the expected normalized cost saving as a result of having a super-query (*Q*) in the cache:

$$PI(Q) = \frac{h(Q)c(Q)}{s(Q)} \quad (2)$$

where *s(Q)* is number of entries for that super-query, *h(Q)* is its hit frequency, and *c(Q)* is the retrieval cost of the super-query, which is assumed to be same for the super-queries belonging to same template. The candidate list is maintained as described below:

For each incoming query a corresponding super-query is added to the candidate list if it is not already present. Otherwise, its hit statistic *h(Q)* is updated. Super-queries which do not have a single hit for the

last N incoming queries are removed from the candidate list. □

The *threshold* is set as $k.PI(Q_L)$ where Q_L is the least popular super-query actually cached. We use the values $k=2$, $N=500$ for our experiments. The popularity index metric is similar *utility value* of greedy dual size popularity (GDSP) algorithm [16].

Compared to query caching, using super-query caching reduces the meta data to be maintained for the same cache size but increases overheads of maintaining statistics to estimate popularity of super-queries. Such prefetching is able to exploit any spatial locality arising out of semantic relation between user queries due to the presence of semantic hot regions.

However, for certain types of queries it is difficult to imagine accesses having hot-regions which can be semantically described by generalized queries. Using super-query caching for such types of queries yields inferior hit-ratio performance and significantly larger overheads compared to query caching. As an example consider a name search based on surnames generating queries `(sn=_*)`. Even if there is a semantic region `(e.g. sn=smith*)` with large number of accesses, the region might not be *hot* since a possibly large number of entries associated with the region brings down its popularity index.

The authors of [13] consider only one type of query, `(telephoneNumber=_)`, which requests an entry with the given telephone number. They generate workloads with between 70-90% queries uniformly distributed inside a hot region and show that super-query caching performs better than simple query caching.

The reason for query caching not performing well in this case can be easily seen. Firstly, generating workloads uniformly within the hot region rules out temporal locality. Secondly, since a maximum of one entry per query is returned, hits are more likely to be from repeat queries rather than contained queries when query caching is used.

From these observations, we infer that the type of query should be considered while making the decision of using query or super-query caching. Since a template specifies the type of query we propose the following template based caching algorithm.

Template based caching
Queries belonging to only a specified number of templates are cached. For each template either query or super-query caching is performed. The cache is dynamically partitioned such that the incremental hits provided by the least recently used queries (query caching) or least popular super-queries corresponding to the same incremental cache size is approximately equal for all the templates. □

The solution was proposed for the disk cache partitioning problem [17] and is applicable to the query cache partitioning problem as well.

6. DIRECTORY SERVER EXTENSIONS

A typical directory server architecture consists of a protocol *front-end* which receives a client request, uses the *backend* to perform the operation corresponding to the request and sends results back to the client. The backend abstracts the database from the front-end by performing the read and write operations corresponding to various LDAP operations.

Most directory servers, e.g. IBM Directory Server, SunOne (iPlanet)/Netscape directory servers support plugins to extend their functionality. Server plugins are libraries including custom functions which can be called before or after an LDAP operation is performed (pre and post operation plugins). Plugins can also be used to integrate a new database with the directory server. Plugins use the *SLAPI API* [21] for interacting with the front end and backend. LDAP query caching can be implemented using a pre-operation plugin for the search operation, called *pre-search* plugin and a database plugin called *proxy backend*. The pre-search plugin implements query containment and other caching algorithms (discussed in Sections 4,5). The proxy backend does not have an associated database and uses the LDAP client API to perform the requested operation on the master server. The cached entries are stored in the *default* database *backend* of the directory server.

The caching algorithms described in Section 5 implement query (or super-query) level cache replacement. When a query is removed from the cache, all the entries which were returned for only that query are removed. To support this a multi-valued attribute `query_id` should be present in each cached entry. The values for the attribute in an entry are the unique identifiers of the queries for which the entry was returned. When a query with identifier, ID, has to be removed, an internal *search* is performed with the filter `(query_id=ID)`. A resulting entry is removed (using an internal *delete* operation) if it has a single value for the attribute, otherwise it is modified (using an internal *modify* operation) to remove the value ID from the `query_id` attribute.

To support weak consistency, queries are assigned a time to live after which they expire and are removed (as described above) from the cache. The time to live can be specified on a per template basis depending on the nature of the queries in the template.

Since the cache consists of a subset of entries from the master server, the DIT at the cache is sparse. This requires the following to be allowed by the default backend: *(i)* Adding an entry without a parent, *(ii)* Deleting an entry with a child, *(iii)* Searching the cache without the *base* in the cache. Also, since the cached entries in general consist of only the required *attributes* in the corresponding query, schema checking (for mandatory attributes) should be disabled while adding or modifying an entry.

Our reference implementation of the directory cache [18] for the OpenLDAP directory server uses the native *slapd* API and a callback mechanism instead of *SLAPI,* which is a recent addition to OpenLDAP [15].

7. APPLICATION OFFLOAD AND PREFETCHING

Application offloading refers to pushing some of the processing from the backend server to a proxy server. For several reasons an offloaded application could be different from the original application. Firstly, it allows cache specific optimizations, e.g. displaying part of a page, which is available from the cache while fetching the rest of the information.

Secondly, it allows prefetching queries based on application logic. Such prefetching can take advantage of the relationship between queries in the same user session. Since these queries might not be semantically related, super-query caching can not exploit this.

Thirdly, in most cases it is not required to offload the complete application but only those components for which interaction with the origin server is not required.

The applications running at the edge are typically much simpler than the backend applications and it is possible to model them using a generic web publishing framework. Modeling applications using such a framework allows separation of content, logic and style and makes it easy to deploy and modify the application.

To model offloaded directory based applications, we use Apache Cocoon [19] which is an XML based publishing framework. An offloaded application is represented as a pipeline of XML-to-XML transformations. A directory application can be modeled in Cocoon using eXtensible Stylesheet Language Transformation (XSLT) and LDAP transformers. XSLT [20] is a language for transforming XML documents. Next we show how the framework can be used to model the offloaded version of a real enterprise directory application.

7.1 Directory Application Modeling: Example

We consider how a real enterprise web based directory application can be modeled using the Cocoon framework. The application has the following states:

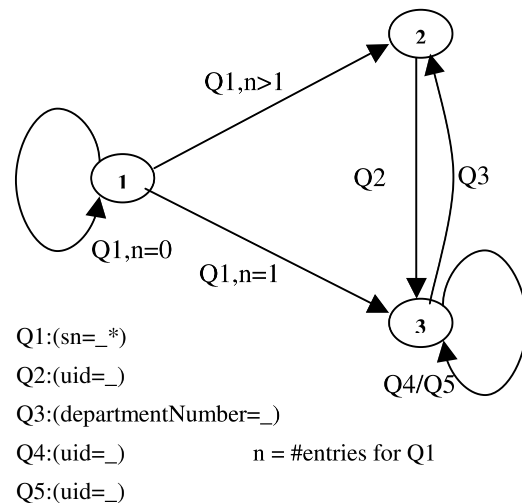

Figure 2: Application State Diagram

State 1: A blank form is presented to the user.
State 2: A list of entries is displayed.
State 3: An individual record is displayed.

In state 1, the user is presented with a form for name search (based on surname). On submitting the input, if more than one entries are returned, a list with links for obtaining details of each entry is displayed (state 2). If a single entry is returned, an individual record is displayed (state 3), which has links to the *manager* and *secretary* entries for an employee and a link to obtain a list of employees in the same department. Clicking on links in state 3 can either result in a list (state 2) or an individual record (state 3). *Q1-Q5* are templates of the corresponding LDAP queries generated.

For each state the user interaction encoded in a URL is an input to a pipeline of transformers. E.g. in state 1 the user input (say `smith`) encoded in a URL is processed by a request generator (R) and converted into an XML representation. An XSLT transformer X_1 maps it to a Directory Services Markup Language (DSML) document representing corresponding LDAP query `(sn=smith*)`. The LDAP transformer (L_1)

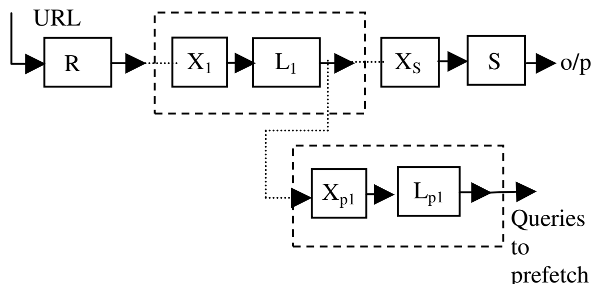

Figure 3: Processing of a user request

performs the corresponding LDAP search operation and converts the result to DSML. Depending upon whether one or more than one entries are returned the XSLT transformer (X_s) uses the results to create an HTML output corresponding to state 3 or 2 respectively which is returned to the client using a serializer (S).

In general the pipeline is of the form $R,X_1,L_1,X_2,L_2\ldots X_n,L_n,X_s,S$ since a request might require a chain of LDAP queries with the result of ith stage providing input for generating LDAP queries for the $i+1^{th}$ stage. For the offloaded application each L_i points to the local LDAP cache.

7.2 Prefetching

The naming model of directory supports entries containing references to other entries using DN syntax attributes. Applications displaying such entries typically include a link to the referenced entries. Similarly web front ends of directory based applications contain links which are indicative of future accesses. Application logic based prefetching can be an important tool to improve hit ratio. Such prefetching can be easily performed using the application offload framework.

Continuing with the example application, if the search results in a single entry, and the accesses indicate a high probability of the *manager* link being accessed, then a new pipeline X_{p1},L_{p1} (Figure 3) can be used to prefetch the manager entry. This ensures that if the manager link is accessed in the next click the LDAP cache will be able to answer the query. In general, a pipeline X_{pi},L_{pi} could be used after each X_i,L_i segment to prefetch LDAP queries at the ith stage.

8. PERFORMANCE OF CACHING ALGORITHMS

We consider the IBM enterprise directory containing more than half a million employee and organizational records for measuring performance of the LDAP caching framework. Each employee entry is approximately 6KB in size. The web based employee white pages application described in Section 7 is the most popular application using the directory. The application is accessed by employees spread across more than 100 countries. The results in this section are based on a workload consisting of more than a million web accesses over a day. The distribution of query-types in the workloads is given in Table 1.

Table 1: Distribution of query types in the workload

Query type	*approx % contribution*
`(uid=_)`	51
`(sn=_*)`	28
`(departmentNumber=_)`	16
Others	5

The directory uses standard LDAP schema for representing common entities. E.g. employee entries use the `inetOrgPerson` object class. The `uid` attribute of this object class is used as a unique identifier for each employee. Query containment for `(uid=_)` queries is equivalent to evaluating the query at the cache and returning TRUE if a matching entry is found. In general, query containment for queries of a template with known number of resulting entries does not require metadata. Thus it is possible to answer those `(uid=_)` queries for which the corresponding entry is cached as a result of some other type of query e.g. `(sn=_*)`. This requires complete entries to be fetched from the master server instead of just the required attributes for all cached queries. For all the algorithms only `(sn=_*)` and `(uid=_)` are considered as cacheable templates, i.e. queries or

super-queries belonging to only these templates are cached. Cache size is measured in number of entries.

a) Performance of query and super-query caching

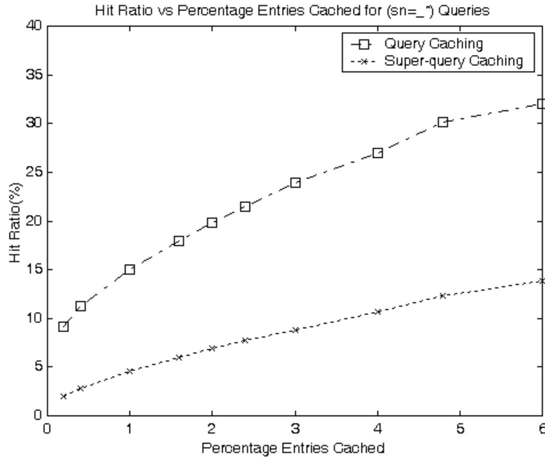

Figure 4: Hit ratio for (sn=_*)

Figure 4 shows hit ratio v/s percentage of master server entries (of class `inetOrgPerson`) cached, for the template `(sn=_*)`, when query and super-query caching are performed. Query caching provides higher hit ratio for a given number of cached entries. A hit ratio of 32% is achieved by caching 6% of the entries. This is due to the significant temporal locality in the access pattern for this template. Super-query caching does not perform well as anticipated in Section 5. Analyzing the hits in query caching reveals two main reasons. Repeat queries account for nearly half the hits. Query refinement (a broad search followed by a restricted one) by users is another important factor.

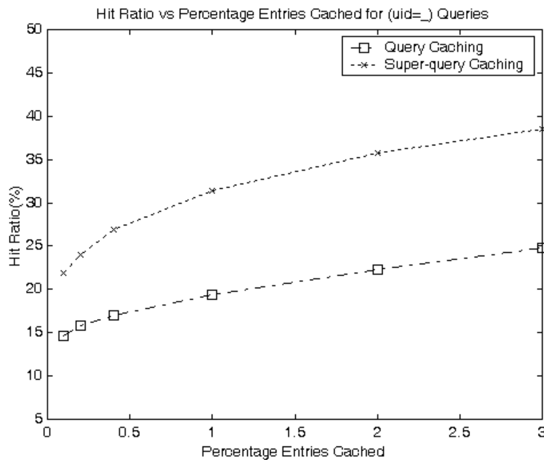

Figure 5: Hit ratio for (uid=_)

Figure 5 shows the comparison for `(uid=_)` template. In this case super-query caching performs better. The access pattern contains spatial locality which is well captured by super-queries. The hit ratio achieved is close to 40% when 3% of the entries are cached.

b) *Template based caching*

Next we present performance of template caching by simultaneously caching different templates with their corresponding best performing algorithms viz. query caching for `(sn=_*)` and super-query caching for `(uid=_)` queries. The performance of template based caching is compared against only query or super-query caching. Figure 6 shows the improvement in hit ratio for template based caching compared to query or super-query caching. Results show that template based caching gives 30-50% more hits than query caching and 15-20% more hits than super-query caching for cache sizes between 1.25-5 % of the directory size.

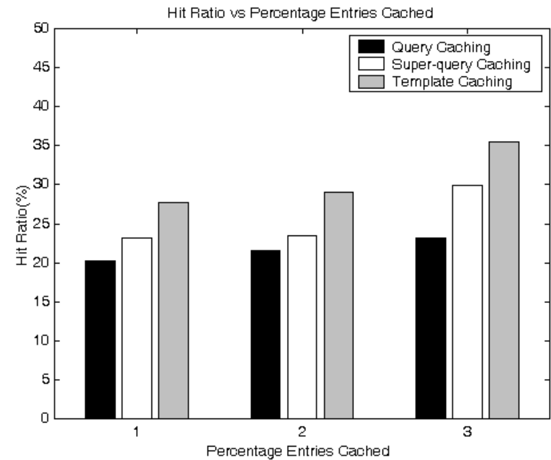

Figure 6: Performance of Template Caching

c) *Effect of application logic based prefetching*

We perform application logic based prefetching of the *manager* entry when an individual record is displayed as described in Section 7.2. This increases the overall hit-ratio by approximately 10%.

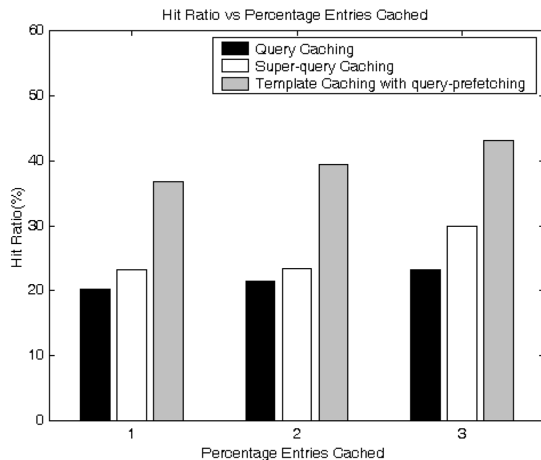

Figure 7: Hit ratio comparison

d) Server load comparison

We measure the server load as the total number of entries served by the master server for the entire length of a workload execution. The workload is executed for the following cases: *(i)* without cache, *(ii)* only query caching, *(iii)* only super-query caching, *(iv)* template caching with application logic based prefetching. Figure 8 shows the server load for cases *(ii)*, *(iii)*, *(iv)* as a percentage of server load in *(i)*. The results show that high gains of template caching with prefetching come at a reasonably low server load.

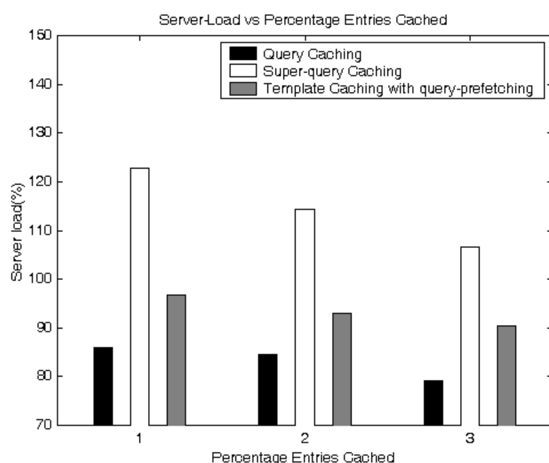

Figure 8: Server load comparison

9. CONCLUSIONS

We consider the problem of improving performance of directory based web applications by caching of LDAP queries. The concept of LDAP templates is defined and is used to reduce the complexity of the query containment problem for queries with *n* predicates to *O(n)*. Caching algorithms which make efficient decisions of caching, prefetching and removing LDAP queries are proposed to improve the cache performance. We have described directory server extensions required in commercial and opensource directory servers to incorporate LDAP caching. Since directories are typically accessed through a web application, we describe means of modeling the offloaded component of the backend application at an edge site. The offloaded application is also used to improve performance of the LDAP cache by prefetching queries based on application logic. The combination of application offload and LDAP caching reduces client latency and improves scalability of the origin directory and application servers.

We evaluate performance of the caching algorithms for a real enterprise directory application using real workloads. The results identify the types of queries which are candidates for query caching or generalized query caching. Results show that for a workload with multiple types of queries the proposed template based caching algorithm provides a 35% hit ratio for a cache size which is 3% of the backend directory and an improvement of 50% over query caching and 15% over super-query caching. We observe that using application logic based prefetching can significantly improve performance by providing up to 45% hit ratio while keeping the server load reasonably low. The algorithms along with the architecture, which supports consistency, security using standard protocols could provide the basis for improving performance and scalability of directory services.

10. REFERENCES

[1] M Wahl, T Howes, S Kille, “RFC 2251: Lightweight Directory Access Protocol (v3)”, www.ietf.org/rfc/rfc2251.txt.

[2] M Wahl, A Coulbeck, T Howes, S Kille, “RFC 2252: Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions”, www.ietf.org/rfc/rfc2252.txt.

[3] T Howes, “RFC 2254: The string representation of LDAP search filters”, www.ietf.org/rfc/rfc2254.txt.

[4] Xin Wang, Henning Schulzrine, Dilip Kandlur, Dinesh Verma, “Measurement and Analysis of LDAP Performance”, International Conference on Measurement and Modelling of Computer Systems, ACM SIGMETRICS, 2000.

[5] Qiong Luo, Sailesh Krishnamurthy, C Mohan, Hamid Pirahesh, Honguk Woo, Bruce Lindsay, Jeffrey Naughton, “Middle-Tier Database Caching for e-Business”, ACM SIGMOD, 2002.

[6] Qiong Luo, Jeffrey F. Naughton, Rajasekar Krishnamurthy, Pei Cao, and Yunrui Li, “Active Query Caching for Database Web Servers”, ACM SIGMOD Workshop on the Web and Databases, WebDB 2000.

[7] Shaul Dar, Michael Franklin, Bjorn Jonsson, Divesh Srivastava, Michael Tan, “Semantic Data Caching and Replacement”, Proceedings of the 22nd VLDB Conference, 1996.

[8] Prasad Deshpande, Kartikeyan Ramasamy, Amit Shukla, Jefferey Naughton, “Caching Multidimentional Queries using Chunks”, ACM SIGMOD 1998.

[9] Qiong Luo, Jeffrey F Naughton, “Form-Based Proxy Caching for Database-Backed Web Sites” VLDB Conference, Rome 2001.

[10] H V Jagadish, Laks V S Lakshmanan, Divesh Srivastava, “Revisiting the Hierarchical Data Model”, IEICE Transactions on Information and Systems, Vol. E00-A, No. 1, January 1999

[11] H.V. Jagadish, Laks VS Lakshmanan, Tova Milo, Divesh Srivastava, Dimitra Vista, “Querying Network Directories”, ACM SIGMOD Conference, Philadelphia, PA, June 1999.

[12] Sophie Cluet, Olga Kapitskaia, Divesh Srivastava, “Using LDAP Directory Caches”, Proc. ACM Principles of Database Systems,1999.

[13] Olga Kapitskaia, Raymond T Ng and Divesh Srivastava, “Evolution and revolutions in LDAP directory caches”, Proceedings of the International Conference on Extending Database Technology (EDBT), 202-216, 2000.

[14] P A Larson and H Z Yang, “Computing Queries from Derived Relations”, Proc. VLDB , 1985.

[15] OpenLDAP Project, web page: (http://www.openldap.org).

[16] Shudong Jin and Azer Bestavros, “Popularity-Aware Greedy Dual-Size Web Proxy Caching Algorithms”, 20th International Conference on Distributed Computing Systems(ICDCS), 2000.

[17] Dominique Thiebaut, “Improving Disk cache Hit Ratio Through Cache Partitioning”, IEEE Transaction on Computers, Vol.41,No.6, June1992.

[18] Apurva Kumar, “The OpenLDAP Proxy Cache” (http://www.openldap.org/pub/kapurva/proxycaching.pdf)

[19] Apache Cocoon, web page: http://xml.apache.org/cocoon

[20] XSL Transformations web page, (http:// www.w3.org/TR/xslt)

[21] Netscape Directory Server: Plug-in Programmer’s Guide (http://enterprise.netscape.com/docs/directory/61/plugin/preface.htm)