Efficient Peer-to-Peer Namespace Searches

Vijay Gopalakrishnan, Bobby Bhattacharjee, Sudarshan Chawathe,Pete Keleher
Department of Computer Science, University of Maryland
$\{$*gvijay, bobby, chaw, keleher*$\}$*@cs.umd.edu.*

Abstract

In this paper we describe new methods for efficient and exact search (keyword and full-text) in distributed namespaces. Our methods can be used in conjunction with existing distributed lookup schemes, such as Distributed Hash Tables, and distributed directories. We describe how indexes for implementing distributed searches can be efficiently created, located, and stored. We describe techniques for creating approximate indexes that can be used to bound the space requirement at individual hosts; such techniques are particularly useful for full-text searches that may require a very large number of individual indexes to be created and maintained.

Our methods use a new distributed data structure called the *view tree*. View trees can be used to efficiently cache and locate results from prior queries. We describe how view trees are created, and maintained. We present experimental results, using large namespaces and realistic data, showing that the techniques introduced in this paper can reduce search overheads (both network and processing costs) by more than an order of magnitude.

1 Introduction

This paper addresses the problem of content-based search in peer-to-peer (P2P) networks. We use indexes on searchable attributes to permit such searches without flooding the network. We also use cached query results, which we call *view caches*, to further improve efficiency. Our work is equally applicable to systems built of either distributed hash trees (DHTs) (e.g., Chord [1], CAN [2], Tapestry [3], and Pastry [4]), or systems built from hierarchical directory services (e.g., TerraDir [5]), but we draw our examples from DHTs in this paper.

A DHT is a distributed and decentralized structure that allows autonomous *peers* to cooperatively provide access to a very large set of data. DHTs use cryptographic hashes to provide near-random association of objects to sites that publish the objects to the rest of the system. An object is looked up by using the hash of the object name to route to the corresponding peer, usually in $O(\log n)$ overlay hops. DHTs provide excellent balance of routing load because paths to a given node vary widely based on the path's origin, and because related objects (even with only slightly different names) are randomly distributed throughout the system.

This random distribution is the strength of the DHT approach, but it also destroys object locality. A set of objects chosen by common attributes or characteristics has its members mapped to peers throughout the system. This distribution, together with the sheer scale and diversity of the source data, makes it impractical to consider solutions that (even periodically) flood the network in order to evaluate queries. Similarly, flooding the network should not be considered a practical primitive when creating indexes designed to aid query evaluation. Instead, such indexes should be created and maintained incrementally. Further, given the nature of DHT-based applications, the indexes should be as distributed and as decentralized as the underlying system.

A straightforward method that satisfies the above requirements is one that maps each index entry to a peer using a hash of the entry's name. Distributing index entries in this manner allows them to be managed using the services provided by the DHTs for data objects. Among these services are transparent caching and replication, which are used to provide fault-tolerance and high availability.

However, the question of efficiency remains. A straightforward use of distributed indexes is potentially quite expensive. DHTs are intended for use with extremely large distributed data sets. (Data sets with up to

10^{14} objects have been discussed [6].) Distributed indexes allow evaluation of single-attribute queries (e.g., `artist="foo"`) without flooding, but a simple conjunctive query with two attributes (e.g., `artist="foo" and genre="bar"`) typically requires the entire index for one attribute to be sent across the network. The indexes may be quite large, as their size may grow linearly with the number of system peers.

We address the shortcomings of the above approach by using *view caching* to efficiently exploit locality in query streams and object attributes. Our work is most applicable to systems in which queries exhibit a high degree of locality. Previous studies (e.g., [7]) have indicated a Zipf-like distribution for queries in P2P systems such as Gnutella. Our approach uses the locality inherent in Zipf-like distributions to cache and re-use results between queries. In particular, our methods maintain not only single-attribute indexes on the searchable attributes (e.g., `artist`, `album`), but also conjunctive multi-attribute indexes, which we call views (e.g., `artist and album`, `artist and album and title and genre`). Our methods permit efficient evaluation of conjunctive queries using these views by using a distributed data structure called a *view tree*. Non-conjunctive queries are evaluated by evaluation of the conjunctive disjuncts in their disjunctive normal forms.

Results presented in Section 4 show that these techniques for evaluating queries can reduce the amount of data exchanged by about 85%. Further, the view tree is a flexible data structure that preserves local autonomy. The decision of what to cache, and for how long, is made locally by each peer, not dictated by a central server or caching discipline.

The main contributions of this paper are as follows:

- We present a distributed data structure (*view tree*) for organizing cached query results (*views*) in a P2P system.
- We describe how to create and maintain multi-attribute indexes while maintaining the autonomy of peers.
- We present methods that use cached query results stored in a view tree to efficiently evaluate queries using a fraction of the network and processing costs of direct methods.
- We present results from detailed simulations and show the benefits of maintaining the view tree.

The rest of the paper is organized as follows. Section 2 discusses prior work. Section 3 presents our search algorithm and our methods for creating and maintaining the view tree. Section 4 summarizes our results. We discuss some issues like index partitioning and aggregation in Section 5. We conclude in Section 6.

2 Related Work

Our work builds on recent work on peer-to-peer namespaces, including DHTs [1, 2, 3, 4, 8, 9, 10, 11], and wide-area directories [5]. The techniques described in this paper can be used to provide indexed search services for all of these systems. There have been a number of other efforts to provide a search infrastructure over peer-to-peer systems. Reynolds and Vahdat [12], Gnawali [13], and Suel et al. [14] discuss search infrastructures using distributed global inverted indexing, where an index is built for each attribute. The model of queries, indexes, and updates proposed in [12] is identical to the model used in this paper: Each index maps a keyword to the set of documents containing that keyword, while each host in the system is responsible for all keywords that map to it. In [12], the authors also investigate how to use Bloom filters to efficiently compute approximate intersections for multi-attribute queries. In this paper, we introduce the idea of result caching using view trees and discuss how these caches can be used to answer queries. We believe that the two techniques are complementary; one could use Bloom filters to compute the intersection of indexes after the set of useful caches have been identified. In [13], the author argues for storing intersections of two keywords instead of storing indexes of single keywords. In [14], the authors propose a two-tier architecture to do P2P full-text searches using inverted indexes.

There have also been proposals to performs searches without global indexes. Tang et al. [15] and Song et al. [16] propose schemes for semantic-based searches over the CAN [2] network. The authors of [15] extend the vector space model (VSM) and latent semantic indexing (LSI) to hash the data to a point in the CAN network. They show that semantically similar data items hash to points that are close to each other in the CAN network. Search is implemented by applying the same technique on the query and flooding the neighborhood until enough results are obtained. The authors in [16] maintain neighbor-lists for semantically similar data in the network. They propagate the message along these neighbor-lists while searching for content. There have also been proposals to build semantic overlays to improve the search in Gnutella-like networks [17, 18]. Annexstein et al. [19] argue for combining text data to speedup search queries, at the expense of more work done attaching/ detaching a peer in Gnutella-like networks. The indexes are kept as suffix trees.

Finally, in [20], Li et al. question the feasibility of Web indexing and searching in Peer-to-Peer systems. They conclude that techniques that reduce the amount of data stored and transferred are required for the idea to be feasible. We show in section 4 that our method reduces the amount of data transferred at the cost of very little disk space and can hence be used as a strategy to make P2P indexing and searching more practical.

3 Searching Large Namespaces

3.1 Data and Query Model

We assume that objects are uniquely identified by their names. Locating an object given its name is the basic operation supported by the P2P infrastructure (e.g., using a DHT). Each object has associated metadata that we model as a set of attribute-value pairs. Searching for objects by querying their metadata is the main operation that we explore in this paper.

Our primary focus is on the distribution, maintenance, and use of indexes and view caches, and not on the methods used for managing individual indexes. Specifically, we are interested in identifying and locating indexes that may be profitably used to evaluate a given query, and not on the specifics of the data structures used for the indexes themselves. Our methods are not dependent on any particulars of the underlying indexing schemes and support equally well the identification and location of profitable B-tree indexes (for range queries) as they do profitable hash indexes (for point queries) or R-tree indexes (for spatial queries). Similarly, our methods could be generalized easily to more structured metadata representations (e.g., XML). To simplify our discussion, however, we assume that attributes are boolean in this paper. Queries are thus boolean combinations of attributes (e.g., $a \wedge (b \vee \neg c)$). Our methods use conjunctive queries of the form $a_1 \wedge a_2 \wedge \ldots \wedge a_k$ as the building blocks for supporting more general queries.

3.2 Attribute Indexes

Since we wish to evaluate queries without flooding the network, we need a method for associative data access. Attribute indexes provide this base functionality by mapping attribute names to objects (object names) with that attribute. Attribute indexes can be stored in the P2P network in a specially designated part of the namespace. For example, the indexes for attributes `manufacturer` and `price` are named `/idx/manufacturer` and `/idx/price`, respectively. (We use `/idx` as the reserved namespace for indexes; in practice, a more obscure name is appropriate.) With this naming scheme, locating indexes is no different from locating other objects in the P2P system. Further, this scheme is applicable to both hash-based and tree-based methods for name-based search. For example, if the index in question is `/idx/price`, the index could be stored in the peer determined by the key $k = \text{Hash}($`/idx/price`$)$. In hierarchical systems like TerraDir, the index is stored in the node with fully qualified name `/idx/price`.

3.3 Materialized Views

While attribute indexes permit single-attribute queries to be evaluated without flooding the network, evaluating multi-attribute queries may require large sets of the object identifiers matching one attribute to be transferred over the network to the sites of the other attribute indexes. To avoid such large data transfers, our method uses cached query results, or *views*. More precisely, a view is the materialized result of a conjunctive query. (Views for non-conjunctive queries are obtained from the disjunctive normal form of the queries.) The idea of using cached results and materialized views to enable faster query processing has beens studied extensively in the database literature (e.g., [21], [22], [23]) and is also related to the problem of answering queries using views (e.g., [24], [25]). However, as we describe below, when views are scattered over a P2P network instead of located at a centralized server, the tasks of view maintenance and query evaluation using these views pose additional challenges.

We now describe the task of locating materialized views that are beneficial to the evaluation of a query. For a given query, this task may be thought of as consisting of two interdependent subtasks. The first task, which we call the *view location problem*, consists of determining the materialized views that exist in the network, preferably restricting our attention to those that are relevant to the query. This task highlights an important difference between this problem and the well-studied problem of answering queries using views in the database literature. Database methods assume that the set of views is well known and typically small, while in a P2P environment this set is generally unknown and large. The second task, the *view selection problem*, consists of determining a good query plan based on the available materialized views (and statistics such as cardinalities of views and selectivities of attributes). Unlike prior work on this problem, we focus on methods that are efficient in a P2P environment without a central repository of view metadata.

At first glance, it may appear that the view location problem can be solved by assigning a *canonical name* to each view (so that equivalent expressions of the same view are unified) and using the facilities of the P2P network for locating the named views. For example, if we render views in canonical form by listing the attribute in sorted order of their names, equivalent views $a \wedge c \wedge b$ and $b \wedge a \wedge c$ both map to $a \wedge b \wedge c$. Unfortunately, this idea solves only part of the problem: locating a view that is equivalent to a given view. For example, it permits us to locate a view $a \wedge b \wedge c$ in response to a request for $a \wedge c \wedge b$, using a canonical name of the form `/idx/abc` (with attribute names sorted lexicographically, in general). However, this idea does not help us locate the views that are helpful in evaluating the above query in the absence of the view $a \wedge b \wedge c$. The number of views that are useful for answering this query is exponential in the size of the query (number of named attributes) even without the repetition of equivalent views. For example, the query may be answered by using views $a \wedge b$ and $b \wedge c$ or using $a \wedge b$ and $a \wedge c$, or using $a \wedge b$ and c, and so on.

Obviously any method that relies on checking the presence of all views that are relevant to a query is not practical. Hence, we must restrict the set of views we consider. Some of these problems could be circumvented by maintaining a centralized list of all materialized views in the network. In a P2P network, this strategy would translate to one peer storing and maintaining all indexes, as well as responding to all index lookups. Such a design has several problems. Not only is it unfair to the index-handling peer, it also creates a single point of failure and produces a performance hotspot.

3.4 The View Tree

Our solution to the above problems is based on a distributed data structure we call the *view tree*. The view tree maintains a distributed snapshot of the set of views materialized in the network, allowing efficient location of the views relevant to a query. Below, we describe the structure of this tree, the search algorithm used to locate views for a query, and the methods used to maintain the tree as both data and views in the network change.

Each node in a view tree represents a materialized conjunctive view. That is, each node corresponds to the cached results of a conjunctive query of the form $a_1 \wedge a_2 \wedge \ldots \wedge a_k$, where a_i are attribute names. For brevity,

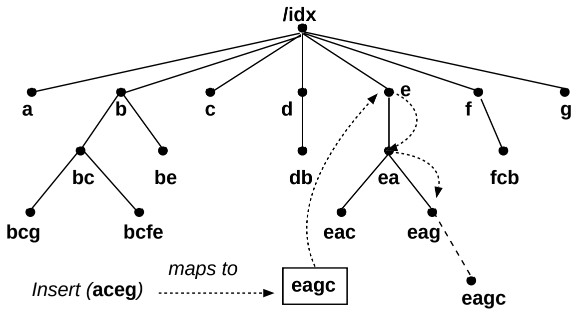

Figure 1: Example of a View Tree. Note that a node $a_1 a_2 \ldots a_k$ represents materialized view $a_1 \wedge a_2 \wedge \ldots \wedge a_k$. The picture also depicts a materialized view $a \wedge c \wedge e \wedge g$ being inserted into the view tree.

we refer to views and their corresponding view-tree nodes as simply $a_1 a_2 \ldots a_k$, the conjunctions being implicit. The *label* $l(n)$ of a node n in the view tree is the view it represents. Each view tree node is, in general, located at a different network host. Thus traversing a tree link incurs a network hop. (Our methods permit several view tree nodes to be mapped to the same host; such aggregation of nodes results in only improved performance.) Figure 1 depicts an example of such a view tree. The root of the tree is labeled with the special index prefix $/\mathrm{idx}$. The first level of the tree represents all the attribute indexes (single-attribute views) in the network. The multi-attribute materialized views are mapped to nodes at greater depths. For example, the node bc corresponds to the view $b \wedge c$ and the node fcb corresponds to the view $f \wedge c \wedge b$.

In order to identify logically equivalent views (e.g., bac, cba, and bca), we refer to each view using a *canonical name*. The simplest choice for such a canonical name is the lexicographically sorted list of attribute names (abc in our example). However, this scheme is likely to result in very unbalanced trees. Subtrees rooted at nodes labeled with attributes that occur early in the sort order would likely to be much larger than those corresponding to attributes later in the sort order. This bias would create routing imbalances and hotspots. We avoid this problem by defining canonical names using a *permuting function* on the attributes of a view. The function is chosen so that it is equally likely to generate any one of the $l!$ permutations of a view with l attributes. (Methods for generating such permutations are well-known [26].)

We may think of the view tree as a modification to a trie over the alphabet of attribute names. In the trie, each node is labeled with a single attribute and the depth of a node equals the number of attributes in the view it represents. The view tree uses a PATRICIA-like scheme in which sibling-less nodes are merged with their parents, concatenating their node labels [27]. Thus each interior node of the tree has at least two children. We define the *residual label* $r(n)$ of a view tree node n to be the suffix of its label that is not included in its parent's label. That is, $l(n) = l(p(n))||r(n)$, where $p(n)$ is n's parent and $||$ denotes concatenation.

We illustrate this scheme in Figure 1. Suppose we want to insert the view $aceg$ into the view tree. Further, suppose that the permuting function applied to this view results in $eagc$. The parent of this view in the view tree is the node corresponding to the longest prefix of $eagc$, i.e., eag. As illustrated by Figure 1, finding the parent of a new view when it is inserted into the view tree is simple: we follow tree links corresponding to each attribute in the view, in order; the node that does not have a link to the required attribute is the parent of the node representing the new view.

3.5 Answering Queries using the View Tree

Given a view tree and a conjunctive query over the attributes, finding the smallest set of views to evaluate the query (using only the views) is NP-hard even in the centralized case, by reduction from *Exact Set Cover* [28]. Thus, an exact solution is not practical, especially in a distributed environment. Not surprisingly, our method for answering queries using views materialized in a P2P network is based on heuristics. However, it possesses the

1: let $q = a_1 \ldots a_m$ {input query}
2: let $c_1, \ldots, c_k$ be the children of n in view tree
3: $q_w \leftarrow q$ {remainder query}
4: $r_w \leftarrow \emptyset$ {useful views}
5: **while** $q_w \neq \epsilon \wedge \exists i, j, s_1, s_2 : s_1 \| a_i \| s_2 = r(c_j)$ **do**
6: $\quad (i^*, j^*) \leftarrow \min\{(i, j) : \exists s_1, s_2 : s_1 \| a_i \| s_2 = r(c_j)\}$
7: $\quad r_w \leftarrow r_w \cup \{r(c_{j^*})\}$ {add child's view}
8: $\quad r' \leftarrow S(c_{j^*}, a_{i^*+1} \ldots a_m)$ {computed at c_{j^*}}
9: $\quad q_w \leftarrow q_w \ominus r'$ {remove newly covered attributes}
10: $\quad r_w \leftarrow r_w \cup r'$ {add recursively found views}
11: **end while**
12: **return** r_w

Figure 2: Search for query q at node n: $S(n, q)$

following desirable properties:

1. *Minimum utility*: If the network contains a materialized view that is logically equivalent to the query, our method is guaranteed to find this view. This property ensures a minimum utility for each view because, at the very least, it benefits future invocations of queries that are logically equivalent to query whose results are cached. Of course, the benefit of materialized views in not limited to such queries. As indicated by our experimental results in Section 4, views are often useful for multiple queries.

2. *Forward progress*: If there is no view equivalent to the query, then each view tree node that is visited by our method results in locating a view that contains at least one new query attribute (one that does not occur in the views located so far). This property not only limits the number of nodes visited to the length of the query (number of query attributes), but also yields tangible incremental benefit at each step of the process, allowing graceful early termination. (Recall that a query for attributes not covered by any materialized views may always be answered by using the attribute indexes, albeit at higher cost.)

Our search algorithm is outlined in Figure 2. Given a query q (in the canonical form described earlier), the algorithm is invoked as $S(t, q)$, where t is the root of the view tree. Recall that $l(n)$ is the view at node n and $r(n)$ part of $l(n)$ that is not in the view of n's parent. We use the notation $s \ominus X$ to denote the string obtained by deleting from s the characters that occur in set X, and $s_1 \| s_2$ to denote the concatenation of strings s_1 and s_2. The computation of $S(n, q)$ at a node n is based on selecting a set of suitable children to which the computation is propagated, recursively. Results from the children indicate the attributes of q that have been covered. As noted earlier, an exhaustive search is impractical. It is easy to verify that the test on line 5 ensures that our method has the exact match and forward progress properties described earlier. The rest of the pseudo-code is concerned with bookkeeping of the attributes of the query that have been covered by the views encountered so far.

Figure 3 depicts the actions of our method for the query $cbagekhilo$. The circled numbers in the figures denote the order in which computation occurs at view tree nodes. (Recall that each view tree node maps to a different network host in general.) Intuitively, the algorithm first locates the best prefix match, which is $cbag$. Even though the longer prefix, $cbage$ is not materialized, the $cbagh$ child of $cbag$ is useful for this query, and thus this node is visited next. The algorithm is now in the forward progress component of the search and proceeds in depth-first manner visiting nodes that represent views that cover at least one uncovered attribute. Also note that node cbe, the query does not proceed to its child $cbeh$ because the node does not have any attribute that has not already been covered. The query can finally be answered using the intersection of the views $cbaghi$, cbe, k and lo.

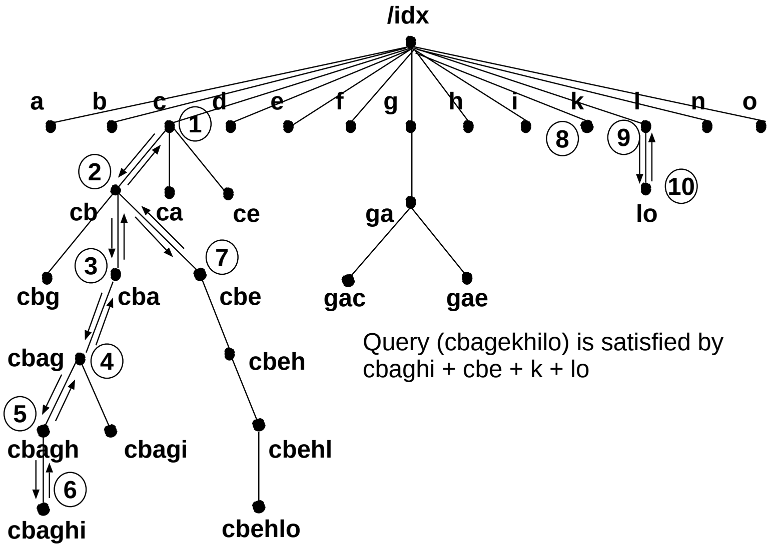

Figure 3: Example of a search using the View Tree. The search proceeds along the direction marked with arrows visiting the nodes in order they are numbered.

3.6 Improving Query Plans

Recall that a query plan as determined by our search algorithm described so far is essentially a cover for the set of attributes in the query using the sets of attributes occurring in the views. We may define an optimal query plan as one that uses views that contain as few tuples as possible, resulting in the smallest data exchange. Given the hardness of even the simpler set-cover problem, insisting on such an optimal query plan is not realistic. However, our preliminary experiments revealed that substantial benefits may be realized by avoiding the plans that fare particularly badly by this metric. In the absence of global data statistics, we estimate the size of a view using the number of attributes in the view definition. Since our queries are conjunctive, views with a larger number of attributes are expected to be smaller, and thus preferred in query plans. We therefore implemented a modified version of our search algorithm that tries to find a query plan in which each view has at least t attributes, where t is parameter that may be set on a per-query basis at runtime. The essential difference from the earlier search method is the stopping condition: the earlier method stops when all attributes in a query are covered. Our modified search method stops when either all attributes in the query are covered with views of length (number of attributes) at least t or when none of the nodes explored so far has a child that represents a view of length l that can be used to replace a view of length $l' < t$ in the current plan.

Figure 4 depicts the actions of the modified method on the query from the previous example. We use a threshold $t = 6$, implying that the method attempts to evaluate the query using views of at least six attributes each. The search proceeds from cbe to $cbeh$ (even though $cbeh$ is not in the canonical order) because the latter covers more attributes. In this process, it also finds $cbehlo$. However, it now starts searching for a six-attribute view that contains k. It does so by visiting every attribute index not yet visited. Since there are no six-attribute indexes containing k, the search visits nodes for all 10 attributes before deciding on using $cbaghi, cbehlo$ and k to evaluate the query. This modification does not have the forward progress property of the earlier method because it may visit nodes that do not cover any additional attributes. However, such nodes result in increasing the length of a view used to cover the query. Therefore, at most $|q| \cdot t$ additional nodes are visited. Typically, the additional query planning work required by this change is small, and is offset by the increased efficiency of query processing resulting from the use of longer views.

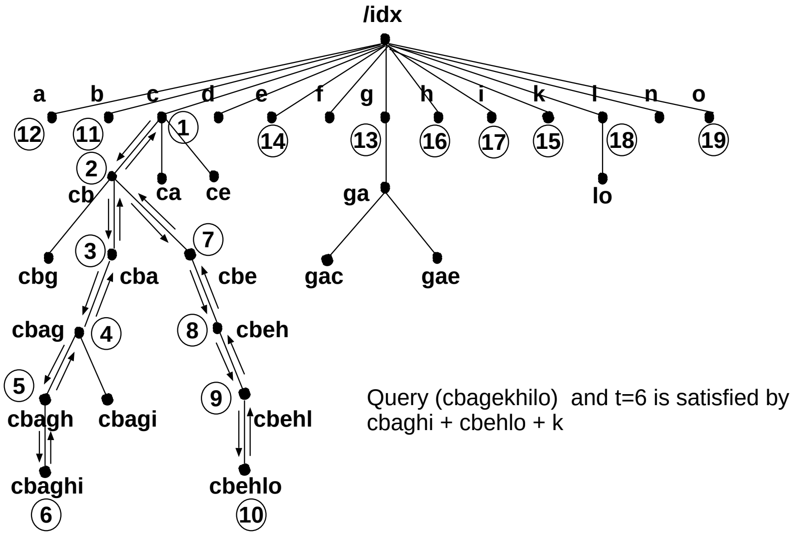

Figure 4: Example of the modified search algorithm. The search still proceeds along the direction marked with arrows visiting the nodes in the order they are numbered.

3.7 Maintaining the View Tree

Since the view tree stores materialized views of network data, there is an implicit consistency requirement: the contents of a materialized view must be identical to the result of evaluating the corresponding query directly on the network data. Thus, the view tree must be modified in response to two kinds of events. The first kind, which we call *data modifications*, are insertions, deletions, and updates of network objects. The second, which we call *view modifications*, are creations and deletions of materialized views, including deletions resulting from the departure of network hosts.

3.8 Data Modifications

When an object is inserted into the network, views that name one or more of its attributes should be updated. This update procedure need not be invoked immediately for most applications. For example, the presence of a document that does not appear in the indexes and views for several hours is not a serious problem for typical text-search applications. Further, not all indexes and views need be updated at once. Since shorter views (fewer attributes) are likely to be used by a greater number of queries, the update procedure may prefer to update them sooner than longer views. Therefore, when an object's insertion is to be reflected in the view tree, we update views in order of increasing length: first, all single-attribute views (i.e., the attribute indexes) over the object's attributes; next, all two-attribute views over the attributes; next, all three-attribute views; and so on.

The procedure for updating indexes and views in response to an object's deletion is completely analogous to the insertion procedure. Updates triggered by deletions can be postponed even longer than those triggered by insertions because the index entries for the nonexistent object can be flagged with a tombstone in a lazy manner when they are dereferenced by applications. Similarly, when a document is updated, we follow the deletion procedure for the dropped attributes and the insertion procedure for the added attributes.

3.9 View Modifications

The policy decisions of which views to cache and for how long are made autonomously by each host in the network using criteria such as query length, result size, and estimated utility. Each host may follow its own

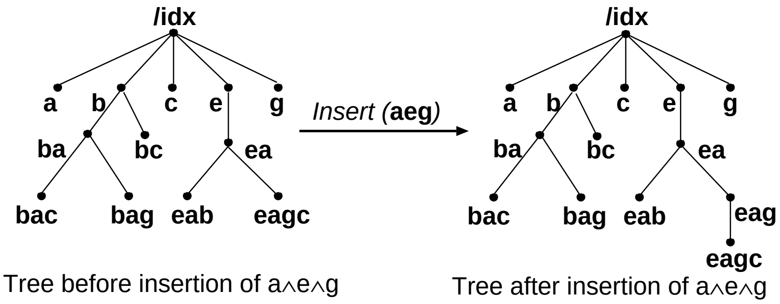

Figure 5: Maintaining the tree when a new node joins.

policy; our methods do not mandate any in particular. For example, a host may decide to cache the results of queries with at most eight attributes, provided each such query contains at least one attribute from a certain hot set and that the corresponding result contains at most ten thousand tuples.

When a view is to be cached, the view tree is searched for the canonical name of the view by following tree branches corresponding to the attribute names in the view, in sequence (as in a standard PATRICIA search). If a match is found, it means that this view was inserted by a concurrent view-insertion operation, and the current insertion is skipped. Serialization occurs at each node on the path from the root of the view tree to the node for the view with a canonical name that is the longest existing prefix of the view being inserted. Necessary changes, such as moving children pointers to the new node, are performed at this time.

Figure 5 depicts the view aeg being added to the tree. The deterministic permutation maps aeg to eag, and the exact prefix ea is found in the tree. The old child of ea, $eagc$, now becomes a child of the newly inserted node eag.

When a view v is to be deleted, we begin by locating its node n in the view tree. If n is a leaf of the view tree, it is simply removed from the tree. If n an interior node, then its parent becomes the new parent of n's children. We also assume that the parent and children exchange heartbeat messages to handle ungraceful host departures. In the case of such failure, the child will attempts to rejoin the tree by following the insertion procedure. Thus, failures result in only temporary partitions from which it is easy to recover.

3.10 Replication

Given that some of the attributes are more popular than others, it is reasonable to expect hotspots among the attribute indexes and the nodes in the view tree. Our system uses the adaptive replication protocol described in (*citation omitted for anonymity*) to handle such hotspots and to prevent the overloading of peers hosting popular indexes. Every peer that holds an attribute index has the ability to request the creation of replicas when its load exceeds a threshold.

Index lookup requests contain data describing the load on the requesting host. While servicing the index lookup request, the receiving host uses this load information to decide if it wants to replicate. Overloaded peers attempt to create replicas in peers that have loads at least 50% lower. Hosts with replicas may create further replicas, resulting in a *tree* of replicas for each index. Figure 6 shows the replica trees for views bag and $eagc$. The numbers in the replica tree represent the server IDs where the view is replicated. The parent of a peer in this tree is the peer from which a replica creation request was received. In the figure, the views at server IDs 613 and 132 created the first replicas and hence are the parents of their respective replica trees.

A peer that receives a replica creation request is not obligated to accept it. If it rejects the request, the requesting peer waits until it finds another suitable candidate. If it accepts the request, it obtains the index data from the requesting peer and sends it an acknowledgment when the indexes have been added. On receiving this acknowledgment, the requesting peer adds a pointer to the responding peer to its list of replicas. This pointer

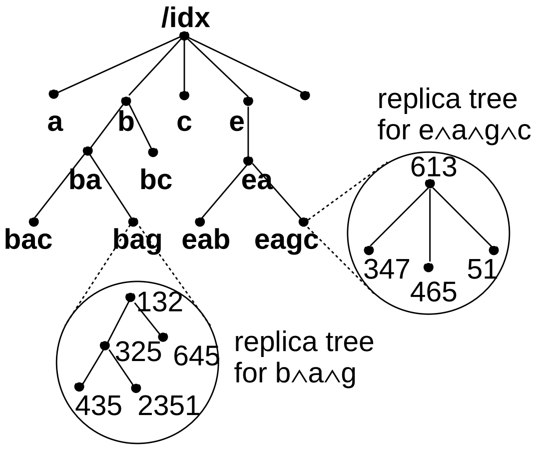

Figure 6: Examples of replica trees.

is also propagated to the requesting peer's parent, if any, in the view tree for this index. When a parent in the view tree receives an index lookup request, it may choose to forward it to one of the replicas for which it holds pointers. Note that replication works orthogonally to the search system. As far as the view tree is concerned, all the replicas of the view are equivalent and can therefore be considered as a single node. Modifications to the index are propagated transparently to all the replicas using the replica tree. When a replicated node is deleted from the view tree, its children are not grafted to its parent as described in Section 3.7; instead, they are left in place because they are reachable from the replicas.

4 Results

4.1 Data Source and Methodology

We chose sets of documents, uniformly at random, from the WT-10g data set from Text REtrieval Conference (TREC) Data as the source data for our experiments. We used HTML pages that exported the `keyword` meta-tag, and used 64 thousand pages for each experiment. The queries were generated as follows. First, we chose a representative sample of Web queries from the publicly available `search.com` query set. Unfortunately, the `search.com` query set does not provide an associated document set over which these queries would be valid. Therefore, we generated queries with the same statistical characteristics as the `search.com` queries using keywords from the TREC-Web data set. Specifically, we used the `search.com` queries to generate the distribution of number of attributes per query. We were also given access to 32 thousand Web queries by the IRCache project and we found that the characteristics of the IRCache and `search.com` queries were comparable. We ran simulations with both data sets with similar results. For brevity, we present results from only the larger `search.com` query set here. We used the distribution of keywords in the set of source documents to map keywords to each attribute. For multi-attribute queries, we generated a set of 50,000 popular keyword digrams, trigrams, etc., and used these, uniformly at random, as the input query set. The popular 50,000 keywords covered all possible multi-attribute queries with non-null results for our source datasets.

We have implemented view trees on both a controlled testbed, and on a detailed packet-level simulator. In our testbed, we have conducted experiments with the all of view tree algorithms, and have tested the replication and tree re-construction protocols. These experiments were conducted on a 40-node testbed, over which we exported and searched 4096 documents selected from the TREC datasets. However, the true benefits of the view tree are only realized on large deployments (hundreds or thousands of servers) of the scale not feasible in our local testbed. We therefore implemented a packet-level simulator to experiment with large system sizes. The simulator and implementation showed remarkable agreement for the small networks that both could run. The results in the rest of this section are from much larger simulated networks (nominally 1000 servers), with simulation parameters as detailed below.

4.2 Simulation Setup

We ran each experiment with 500,000 queries; this number was sufficient in all experiments for the caching behavior to stabilize. For each experiment, we use a *working set*, which is a set of unique queries from which some fraction of the overall queries are drawn. We used a working set of size 50,000, from which 50%, 90%, or 99% of the queries are constructed. We used the 90%-to-working-set stream of queries as representative of a Zipf input. Henceforth, we refer to this stream as the *Zipf input*. All queries, including the queries in the working set, were generated using the scheme described above. However, 90% of the queries were always directed to the queries in the working set, while 10% were unconstrained.

The base system for our experiments consisted of 1000 servers, exporting 65,535 data items and a maximum of 15 attributes per data item. The query inter-arrival time was exponentially distributed with an average of 10 milliseconds. We assume that 10% of servers (approximately 100) fail over the period of all the experiments, unless otherwise noted. We also assume that a server that fails does so ungracefully; i.e. it does not inform peers about its departure. We further assume that a failed server does not recover during the simulation. The upper and lower thresholds for replication are set to 0.75 and 0.3 respectively, while the capacity of each server is assumed to be 10 queries/second. Obviously, this is an artificially low value, but we had to resort to such scaled down capacities in order to reduce the running time of the experiments.

We assume that hosts allocate disk space for the caches separately from the disk space for the attribute indexes. In our experiments, we noticed that single attribute indexes have 600,000 tuples all together. Since we use 1000 servers, a uniform distribution of data would lead to 600 tuples per server for holding the indexes. In practice, the distribution is nonuniform and some hosts may have to allocate more than this number to hold the attribute indexes. Since we are only interested in the space taken by the caches we create, the disk space used by the attribute indexes is not important in our experiments. However, we allocate disk space in multiples of 600 tuples for our caches. Specifically, unless otherwise mentioned, we assume that disk space for 600 tuples is set aside for the view caches at each server. Once again, in actual deployment, we expect the disk capacities of servers to be much larger as well.

4.3 Evaluation Metrics

The view tree algorithm described in this paper is specifically targeted towards efficiently answering multi-attribute queries. In fact, as long as the single attribute indexes are kept current, the view tree is not used for any single attribute query. As shown in Figure 7, 36% of our queries are single attribute queries, and these were faithfully executed in our simulator. However, the contributions of this work pertain to multi-attribute queries, which we evaluate using three metrics:

- *Number of tuples intersected* - Our primary metric is the number of tuples intersected when evaluating multi-attribute queries. There are two points to note:
  - In our results, we report the number of tuples transferred between peers for query evaluation, and not the number of tuples transferred as the result to the query. Larger indexes will have to be intersected when a view tree is not used, since multi-attribute indexes, which are inherently smaller, are not available.

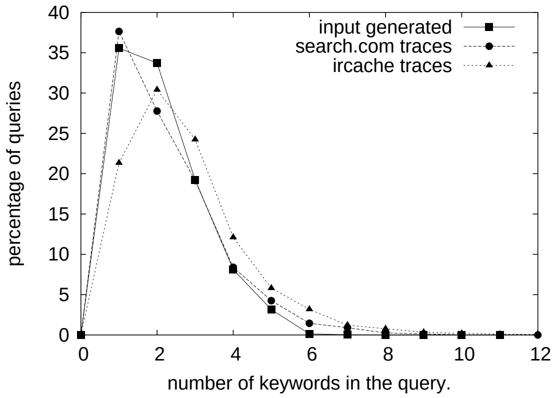

Figure 7: The distribution of keywords in queries in our generated input and in traces obtained from search.com and IRCache.

For our data, the average sizes of the different indexes are shown in Table 1. If an exact result is required, then the number of tuples intersected is also an upper bound on the number of tuples that must be transferred between hosts. Thus our metric represents a worst case for the view tree.

– If an approximate method (e.g., [12]) is used, then it is not necessary to transfer large indexes over the network. It is sufficient to transfer the approximate indexes (e.g., Bloom filters) to a single location where the intersection is computed. However, the number of tuples that must be processed is still equal to the number of intersected tuples. Thus, this metric also provides a lower bound on the amount of processing that must be done by hosts in the search network to answer a query.

- *Storage cost* - We report the number of tuples in the single- and multi-attribute indexes stored in the network. Once again, the particular storage requirements are entirely a function of the input data set and, in our experiments, conform to the averages shown in Table 1. The tuples in the single-attribute indexes must be stored if exact results for these attributes are required without flooding the network. However, the multi-attribute indexes are not required for correctness. We explicitly account for the storage required for multi-attribute indexes and experiment with different replacement strategies for these indexes.

- *View maintenance overhead* - We also model updates to the data. Specifically, we consider attribute insertion, deletion, and update. New attributes, for both insertion and updates, are chosen using the original keyword distribution generated from the complete source data set. For deletion, attributes are selected uniformly at random. Both single- and multi-attribute indexes have to be updated as data items are inserted, updated, or deleted from the namespace. We present results quantifying the number of messages that are required to update the attribute indexes. We specifically account for the number of messages that are sent directly as a result of maintaining the view tree. Once again, we assume the worst case scenario in which each index is hosted by a different host. Thus, the overheads we present represent an upper bound on the number of messages and tuples that would have to be transferred in a deployed system.

There are a number of other metrics we have analyzed that are of academic interest. They do not intrinsically account for overheads or performance, but instead help us understand the system better. These include number of indexes created, multi-attribute index hit rates, and index replacement rates when storage bounds are enforced. We instrumented our simulator to log these numbers as well, and report them below.

# of attributes	90% locality			Random Queries		
	min	avg	max	min	avg	max
1	1	29.38	6658	1	29.38	6658
2	0	33.39	3139	0	7.85	3139
3	0	26.55	1942	0	0.31	1118
4	0	2.73	1918	0	0.01	111
5	0	3.91	753	0	0.00	2
6	0	0.00	0	0	0.00	0

Table 1: The minimum, average, and maximum sizes (number of tuples) of n-attribute indexes, when 90% of the queries are from a set of 50,000.

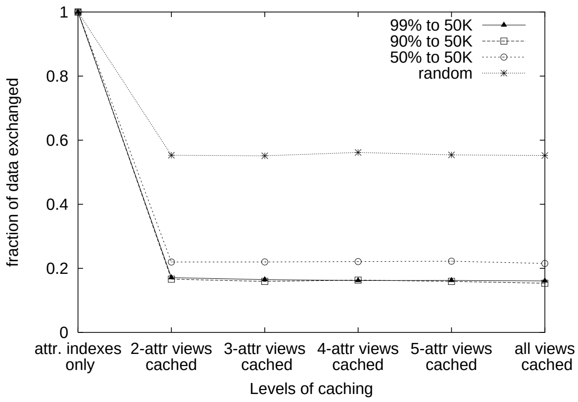

Figure 8: Caching benefit by level.

4.4 Utility of View Caches

Our first set of experiments studies the benefit of maintaining the view tree. These experiments used our base system and 500,000 queries. We ran experiments with all the query streams: 50%, 90%, and 99% generated from a working set of 50,000 queries, and a random query stream. Figure 8 depicts the number of tuples transferred for query evaluation, for different levels of caching for all the query streams. The y-axis depicts the normalized number of tuples transferred for each level of caching indicated on the x-axis. The y-axis is normalized by setting the number of tuples transferred for the single-attribute indexes to one. The number of tuples actually transferred for the single attribute indexes was 60.89M for the random queries, 227M for the 50% case, 437.31M for the 90% case and 445.52M for the 99% case.

The figure illustrates the benefits of maintaining a view tree: maintaining only the 2-attribute indexes reduces the number of tuples transferred for the skewed inputs by 82% (from 437.3M for attribute indexes only to 72.9M tuples for caching 2-attribute indexes). Caching all the views further reduces the intersection overhead to about 85% of the original (from 437.3M to 67.4M). The same trend is repeated for all other skewed query streams as well.

Figure 8 also indicates that even the random set of queries benefit from the view tree, with a 45% reduction in the number of intersected tuples. This result may be counterintuitive, since there is no locality in the queries, but is explained by the following: if two of the requested attribute indexes are large and their intersection is small, then storing the result in a cache can reduce the data that is transferred while computing any new query that requires these two attributes. As Table 1 indicates, for the random case, the average size of indexes with two or more attributes is much lower than that of single-attribute indexes.

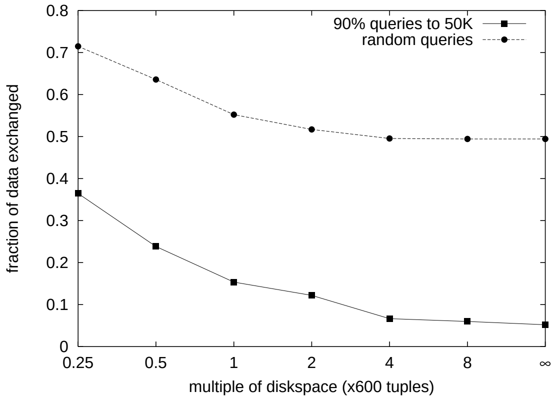

Figure 9: Effect of disk space: Even a small fraction of space allocated for the caches helps reduce the number of tuples intersected.

4.5 View Maintenance: Updates

# of Updates	Zipf queries: # of Extra Updates	Zipf queries: Reduction in data transferred	Random queries: # of Extra Updates	Random queries: Reduction in data transferred
500 K	469.7 K	349.55 M	390.6 K	29.26 M
50 K	82.82 K	403.55 M	62.43 K	30.60 M
5,000	9,078	409.66 M	5,840	30.74 M
500	877	410.19 M	674	30.89 M

Table 2: Update overhead for 500,000 queries. All the results are in number of tuples. Zipf refers to 90% of the queries directed to the 50K working set.

Table 2 summarizes the cost and benefit of maintaining the view tree. In these experiments, we cache all views (which corresponds to the worst case for update overheads) and vary the number of updates. In each update, an attribute is added to or deleted from an object. We perform one update for every k queries where $k = \{1, 10, 100, 1000\}$. The number of actual updates performed depends on the number of caches in the system and the replicas of these caches. In the table, we present (1) the number of *extra* updates that have to be performed to maintain the view tree, and (2) the reduction —due to the view tree— in the amount of data transferred to answer the queries. From the table, it is clear that even for unrealistically high update rates (one update per query), the view tree update overhead is essentially negligible (about 1% in the very worst case).

Note that in general, the number of updates is higher in the skewed query stream. This is because each update has to propagate to more caches and replicas. Of course, as is evident from the table, this small increase in number of updates is quickly dwarfed by the savings due to the cache hits in the query phase.

4.6 Effect of Disk Space

In this experiment, we quantify the effect of the amount of disk space caches on the efficacy of the view tree. Recall that in our simulations, the nominal amount of disk space is 600 tuples. Assume k denote this unit (600 tuples), and in Figure 9, we show how well the view trees work for disk space allocations of $k = \{0.25, 0.5, 1, 2, 4, 8\}$. The figure also contains a result with no replacements (unbounded disk space) ($k = \infty$).

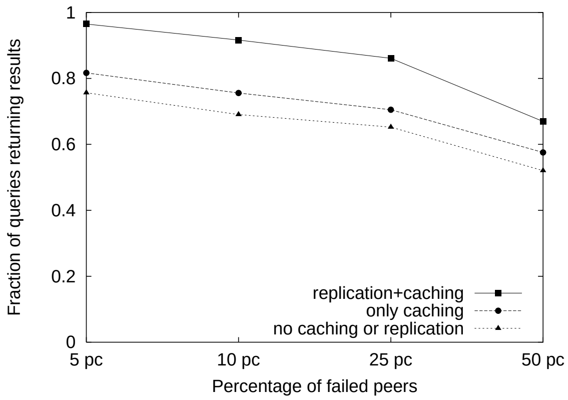

Figure 10: Query processing with server failures.

As expected, the number of tuples exchanged to resolve a query reduces as the amount of disk space increases (because there are more direct cache hits). With locality in the query stream, the number of tuples exchanged drops to 36% with only $k = 0.25$. Also, for our experiments, at $k = 8$ the performance is essentially equivalent to having unbounded space. Lastly, we note, these results are from experiments performed with both replication (which increases disk space requirements) and server failures (which decrease the total available disk space).

4.7 Effect of Server Failures

Recall that we do not assume that a failing server updates state information. Instead, its peers discover the failure independently and update their own states. We performed experiments to analyze the effect of result caching and adaptive replication on reliability in the face of server failures. We used no caching and no replication as the base case and used a Zipf query stream. In each run, the number of failing servers ranged from 50 to 500. As Figure 10 illustrates, result caching increases the number of successful queries (by about 35,000). With the addition of adaptive replication, we see improvements over the base case by as much as 20% when there are 50 failures and 15% when 500 servers fail.

Figure 11 summarizes our experimental results on the effect of adaptive replication and view caching on the number of queries successfully answered. Here, 500K queries come into the system of 1000 servers, with 100 servers failing over time. As the graph shows, the replication scheme increases the number of queries answered from 69% to 91% when only attribute indexes are stored, and from 76% to 92% with all views cached. However, the scheme does not perform very well with random queries, at times performing a little worse than the no replication case. This result is due to the fact that when there is query locality, the load on a server is due to the popular indexes and hence they are replicated. When there are failures, this replication increases the chances that a query to these indexes is answered. Locality also explains the increase in the number of successful queries with no replication. When there are failures, the greater the amount of search required, the higher the probability of not finding an index. Caching views reduces the amount of search needed to answer queries with locality. The fluctuations in the replicated case are due to indexes having to be replaced to accommodate the replicas in the available disk space. In this process, some caches of unpopular keywords are lost and hence queries to to the attributes in those caches cannot be evaluated. Note that this happens only when the single attribute indexes corresponding to the requested attributes are also lost due to server failure.

In the random case however, replication does not necessarily replicate the requested index, and so does not help availability. On the contrary, replication might lead to the replacement of the requested index from the cache.

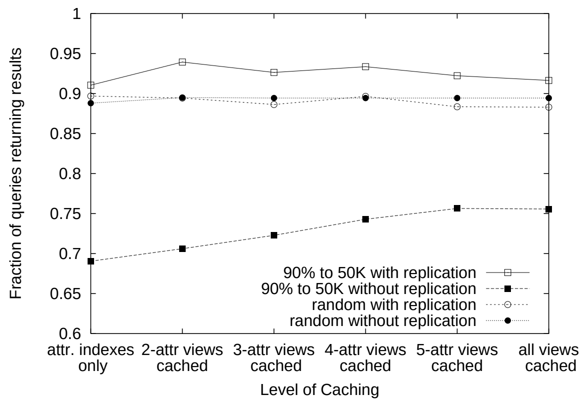

Figure 11: Adaptive replication helps answer more queries when there is locality in the query stream.

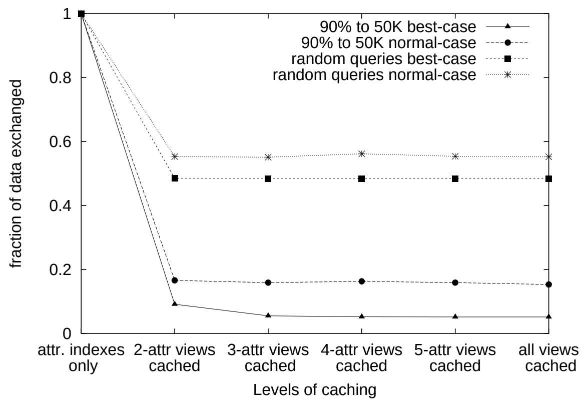

Figure 12: Comparison of Performance under best- and normal-case.

4.8 Upper Bounds on Query Performance

In order to further quantify the effect of server failures and limited disk space on query performance, we compared our results with those for a network with no failures and unbounded disk space (labeled *best case*). We used two query streams: random and Zipf. We also varied the maximum number of attributes in a cached view for each run.

Figure 12 summarizes the results. In the best case, 2-attribute views and a Zipf query distribution results in approximately 90% reduction in data transfers. This number goes up to 95% when we store all the materialized views. A more realistic system, with 10% of servers failing and diskspace equivalent to 600 tuples, still provides significant benefit (e.g., 85% reduction for the Zipf input when we cache all the views).

5 Discussion

5.1 Limiting the fanout in a hierarchical system

In a hierarchical system, that uses tree structures for locating objects by name (such as the TerraDir), a naive application of the index-location method is likely to cause performance problems due to overloading of the `/idx` node, which has one child node for each attribute known to the network. A more pragmatic approach, and one we use, is to limit the fan-out of the `/idx` node and other nodes in the indexing namespace. A number of data structures suggest themselves for this purpose. For example, a structure such as the *dB-tree* described in [29] can be used to limit fan-out and maintain a balanced tree (thus bounding lookup cost). We may also use methods based on distributed linear hashing, such as LH* [30]. Note that this problem does not afflict the hash-based schemes.

In our current implementation, we use a simpler method. Initially, attribute indexes are inserted in the namespace using the names suggested above. After the fanout of the `/idx` node reaches a threshold (which need not be fixed in advance), index insertions result in the creation of nodes at depth greater than one in the `/idx` namespace. We may describe the change as follows: A new index node is temporarily added to the namespace as a child of `/idx`. The namespace `/idx/*` is then searched for a set of nodes that share a longest common prefix. A new node labeled with this prefix is inserted as a child of `/idx` and parent of the group of indexes. A similar procedure is used to insert additional interior nodes in the namespace whenever the fanout of a node gets too large. (Nodes may use different thresholds or other methods to determine the limits on fanout.)

The above method for inserting interior nodes into the `/idx/*` namespace separates a set of nodes that share a longest common prefix. This choice is motivated by the desire to split off as large a portion of the index namespace as possible, without incurring major restructuring costs. An alternative scheme is to separate a set of nodes of largest cardinality, in order to reduce the fanout by as large an amount as possible in one step. Of course, we could also use a balanced tree data structure, but our experiments (Section 4) suggest that this simple scheme suffices.

5.2 Partitioning Indexes

So far, we have assumed that all attribute indexes are small enough for each replica to fit on a single host. For some applications, this assumption may not hold. For example, in a P2P full-text search environment, the inverted file indexes for commonly occurring words are likely to be several gigabytes in size. It is possible to cope with large indexes by provisioning capacity at a few nodes. However, such a solution reduces the flexibility of the P2P system and we do not consider it further here. Instead, we cope with large indexes by splitting them into components that are small enough to be stored on single hosts.

An index is split by logically partitioning the namespace and storing the index entries in each partition separately. For example, in a DHT environment, we can split index entries using the prefix of the ids. The partitions of an index may be further partitioned (e.g., by splitting using other bit positions). The method for managing the partitions of an index is very similar to the method used for managing replicas. Just as replicas may spawn their own replicas, index partitions may be further partitioned. Pointers to index partitions are treated analogously to pointers to replicas, with one difference: While a lookup is forwarded to only one replica, it must be forwarded to all partitions of an index.

5.3 Index Availability

If the index for an attribute were to become unavailable due to node or network failures, the performance of queries containing the attribute would be greatly hurt. In the worst case (e.g., a query that searches only on the attribute whose index is unavailable), it would be necessary to flood the network in order to generate complete results. Such flooding is not required for all queries that mention the attribute whose index is unavailable. For example, if the index on a fails, the query $a \wedge b \wedge c$ can be evaluated by first evaluating the query $b \wedge c$ and then checking each object in the result for a. Although there is no flooding in such cases, a substantial amount of additional work is incurred because a potentially large number of objects must be retrieved and checked for the attribute a. When objects are substantially larger than names (e.g., multimedia objects or large XML files), retrieving objects in this manner for query processing adds a large amount of network traffic.

Thus, it is essential that indexes (especially those that require flooding to recreate) be statically replicated, and that these replicas be kept up to date when the original index is updated. The number of static replicas needed depend on the probability of simultaneous node failures, but in most cases, we expect two static replicas should suffice (this assumes that three nodes, chosen uniformly at random, do not fail in the system within the recovery period). Once the requisite number of static replicas are created, they can use standard techniques (e.g. based on heartbeats [31]) to ensure that the proper number of replicas always exist in the system. Note that the locations of the current set of static replicas for a (partial) index must be consistently maintained at all other replicas.

An interesting issue is the form in which the static replicas store the index data. The most obvious (and often optimal) approach is to simply keep a literal copy of the index data at each replica. Note however that the static replicas are only used for failure recovery (and not for answering queries — separate load-based replicas are used for that), and it is possible to use erasure coding, e.g. Reed-Solomon codes [32], to reduce the amount of storage required for keeping replicas. (It is also possible to use newer probabilistic codes, such as Tornado [33] and LT- codes [34], without changing the essential argument).

Specifically, suppose there are N nodes in the system and there are and average of F failures, each occuring independently with the probability f. In the event that an index is split, it has m partitions. In the event that replication is employed, R replicas are created. In the event that erasure codes are employed, we assume that each block is split into k fragments, to generate n fragments of which any k are sufficient to re-generate the index. We are interested in the probability that we are not able to answer a query in the event of failure.

- Consider the case when the index is not partitioned and is not replicated. Then the Pr[not answering a query] = Pr[index not existing]. This is given by F/N. Therefore

$$Pr[\text{not answering a query}] = \frac{F}{N} \tag{1}$$

- In case we have replication, then the query cannot be answered if all the replicas die. Also, this is true when $R \leq F$. When $R > F$, we will always answer the query.

$$Pr[\text{not answering a query}] = \left(\frac{F}{N}\right)^R \tag{2}$$

- If the index is split into m partitions, then the query cannot be answered is one or more partitions are un-available. We can verify that this turns out to be,

$$Pr[\text{not answering a query}] = 1 - \left(1 - \frac{F}{N}\right)^m \tag{3}$$

- If the index is split into m partitions and each partition is replicated R times, then we can verify that the probability of not answering a query is $1 - \left(1 - \left(\frac{F}{N}\right)^R\right)^m$. This can be shown easily.

$$\begin{aligned}
Pr[\text{not answering a query}] &= Pr[\text{some block(s) does/do not exist}] \\
&= 1 - Pr[\text{all blocks exist}] \\
Pr[\text{a block exists}] &= Pr[\text{some replica of the block exists}] \\
&= 1 - Pr[\text{none of the replicas exist}] \\
Pr[\text{a replica does not exist}] &= \frac{F}{N}
\end{aligned}$$

$$
\begin{aligned}
Pr[\text{none of the replicas exist}] &= \left(\frac{F}{N}\right)^R \\
\therefore Pr[\text{a block exists}] &= 1 - \left(\frac{F}{N}\right)^R \\
\therefore Pr[\text{all blocks exists}] &= \left(1 - \left(\frac{F}{N}\right)^R\right)^m \\
\therefore Pr[\text{some block does not exist}] &= 1 - \left(1 - \left(\frac{F}{N}\right)^R\right)^m \qquad (4)
\end{aligned}
$$

- Consider the case when we have erasure codes. If the index is split into m partitions and each partition is encoded into n fragments of which k are sufficient for re-generating the block. In this case,

$$
\begin{aligned}
Pr[\text{not answering a query}] &= Pr[\text{some block(s) is(are) not available}] \\
&= 1 - Pr[\text{all blocks are available}] \\
Pr[\text{a block is available}] &= Pr[\text{more than k fragments are available}]
\end{aligned}
$$

$$
\text{let } X_i = \begin{cases} 0 & \text{if the } i^{th} \text{ fragment is lost} \\ 1 & \text{if the } i^{th} \text{ fragment is available} \end{cases}
$$

Let $f = F/N$ be the probability of failure.
Let X be the random variable denoting the number of fragments that are available. Then

$$
X = X_1 + X_2 + X_3 + \ldots + X_n
$$

We want $P(X \geq k)$.

$$
\begin{aligned}
P(X \geq k) &= 1 - P(X < k) \\
P(X < k) &= P(X = 0) + P(X = 1) + \ldots + P(X = k-1) \\
&= f^n + \tbinom{n}{1} \cdot (f)^{n-1} \cdot (1-f) + \ldots + \tbinom{n}{k-1} \cdot (f)^{n-k+1} \cdot (1-f)^{k-1}
\end{aligned}
$$

Since X is the sum of independent indicator random variable, we can apply Chernoff bounds to this equation. By Chernoff bounds,

$$
P(X < k) < \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu} \text{ where } \mu \text{ is the mean and } k = \mu(1-\delta)
$$

$$
\therefore \mu = n \cdot (1-f) \text{ and } \delta = 1 - \frac{k}{n \cdot (1-f)}
$$

Since $\delta \rightarrow 1$ for most of the useful values of k and n, we can approximate the above to

$$
\begin{aligned}
P(X < k) &= e^{-n \cdot (1-f)} \\
\Rightarrow P(X \geq k) &\geq 1 - e^{-n \cdot (1-f)} \\
\therefore Pr[\text{all blocks are available}] &\geq \left[1 - e^{-n \cdot (1-f)}\right]^m \\
\therefore Pr[\text{not answering a query}] &< 1 - \left[1 - e^{-n \cdot (1-f)}\right]^m \qquad (5)
\end{aligned}
$$

5.4 Approximate Indexes

We describe a technique for creating approximate indexes which can be used by individual nodes to aggregate index state from multiple distinct indexes. An *approximate index* is used to locate data with a given set of attributes. However, unlike a precise index, the approximate indexes may not directly point to the set of data items that have the specified attribute; instead, an extra post-processing step must be employed at run-time to fully resolve any given query. The post processing step may be in the form of a set of directed queries in a constrained portion of the namespace, or may require filtering useful data from a superset of the tuples that match the requisite set of attributes. However, the approximate indexes consume far less space, and are an essential mechanism for efficiently available resources, especially in cases when the number of attribute indexes is very large, and there are not enough resources in the system to store all attribute indexes.

Consider a hierarchical namespace, and let server S store an index I_A for attribute A. Let data items $/a/b/c/d/e$, $/a/b/c/d/h$, and $/a/b/c/d/f$ be part of index I_A, and assume that server S wants to aggregate this index. Instead of storing state about each item individually, node S may aggregate items $/a/b/c/d/[e, f, h]$, as $/a/b/c/d/\star$. When this index is used, a sub-query under the $/a/b/c/d$ must subsequently be resolved for attribute A in order to find the individual data items that belong to the index. In general, a server can aggregate along the namespace by storing a sub-tree pointer for sets of items with common predecessors. The level of aggregation is be controlled by finding predecessors "higher up" in the namespace. Such aggregation along the namespace trades off accuracy versus storage, but still provides an efficient mechanism searching using attributes.

In a flat namespace, e.g. a DHT, it is not possible to aggregate along the namespace. Instead, we must aggregate along different axes. For example, it is possible to take two attribute indexes, say I_A and I_B, for attributes A and B, respectively, and store a single index for attribute $A \vee B$. Any query for attribute A is satisfied using this "unified" index which provides a superset of the set of data items that have attribute A. In order to completely satisfy the query for attribute A, a post-processing step must be performed in which each item in $I_{A \vee B}$ is further checked for attribute A.

6 Conclusion

We have described the design of a text/keyword search infrastructure that operates over distributed namespaces. Our design is independent of the specifics of how documents are accessed in the underlying namespace, and can be used with all DHT-like P2P systems. Our main innovation is the view tree, which can be used to efficiently cache, locate, and reuse relevant search results. In this paper, we have described how a view tree is constructed and updated, and how multi-attribute queries can efficiently be resolved using a view tree. We have also described how individual views can be replicated to handle hotspots and overloads, and have described techniques for reconstructing the tree upon failures.

Our results show that a view tree offers significant benefits over maintaining simple one-level attribute indexes. For trace data, the view tree reduces multi-keyword query overheads by over 80%, while consuming little resources in terms of network bandwidth and disk space. Our results show that a view tree permits extremely efficient updates (essentially zero overhead), and can produce significant benefits when servers fail in the network. Overall, our results are compelling, and show that view trees efficiently enable much more sophisticated document retrieval on P2P systems than is currently feasible.

References

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[2] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, "A scalable content-addressable network," in *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[3] Ben Zhao, John Kubiatowicz, and Andrew Joseph, "Tapestry: An infrastructure for wide-area location and routing," Tech. Rep. UCB//CSD-01-1141, U.C.Berkeley, Berkeley, CA, April 2001.

[4] Anthony Rowstron and Peter Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proceedings of IFIP/ACM Middleware 2001*, Heidelberg, Germany, November 2001.

[5] Bobby Bhattacharjee, Peter Keleher, and Bujor Silaghi, "The design of terradir," Tech. Rep. CS-TR-4299, University of Maryland, College Park, MD, October 2001.

[6] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proc. of the* 9^{th} *International Conference on Architectural Support for Programming Laguages and Operating Systems*. ACM, November 2000.

[7] Kunwadee Sripanidkulchai, "The popularity of gnutella queries and its implications on scalability," Available from `http://www.cs.cmu.edu/~kunwadee`, February 2001.

[8] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse, "Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead," in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, March 2003.

[9] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman, "Skipnet: A scalable overlay network with practical locality properties," in *Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, March 2003.

[10] Dahlia Malkhi, Moni Naor, and David Ratajczak, "Viceroy: A scalable and dynamic emulation of the butterfly," in *21st ACM Symposium on Principles of Distributed Computing (PODC '02)*, Monterey, CA, August 2002.

[11] M. Frans Kaashoek and David R. Karger, "Koorde: A simple degree-optimal distributed hash table," in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, March 2003.

[12] P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," Available from `http://issg.cs.duke.edu/search/`.

[13] Omprakash D Gnawali, "A keyword set search system for peer-to-peer networks," M.S. thesis, Massachusetts Institute of Technology, June 2002.

[14] T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram, "Odissea: A peer-to-peer architecture for scalable web search and information retrieval," in *6th International Workshop on the Web and Databases (WebDB)*, June 2003.

[15] Chunqiang Tang, Zhichen Xu, and Sandhya Dwarkadas, "Peer-to-peer information retrieval using self-organizing semantic overlay networks," in *Proceedings of ACM SIGCOMM '03 Conference*. 2003, pp. 175–186, ACM Press.

[16] Jiantao Song, Yong Zhang, and Chaofeng Sha, "Building semantic peer-to-peer networks upon can," in *In Proceedings of Fifth International Workshop on Networked Group Communications(NGC)*, Munich, Germany, September 2003.

[17] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang, "Efficient content location using interest-based locality in peer-to-peer systems," in *22nd Annual Joint Conference of the IEEE Computer and Communications Societies(INFOCOM 2003)*, San Fransisco, CA, April 2003.

[18] Edith Cohen, Amos Fiat, and Haim Kaplan, "Associative search in peer to peer networks: harnessing latent semantics," in *22nd Annual Joint Conference of the IEEE Computer and Communications Societies(INFOCOM 2003)*, San Fransisco, CA, April 2003.

[19] F. S. Annexstein, K. A. Berman, M. Jovanovic, and K. Ponnavaikko, "Indexing techniques for file sharing in scalable peer-to-peer networks," in *Proc. the* 11^{th} *IEEE International Conference on Computer Communications and Networks*, oct 2002.

[20] Jinyang Li, Boon Thau Loo, Joseph Hellerstein, Frans Kaashoek, David Karger, and Robert Morris, "On the feasibility of peer-to-peer web indexing and search," in *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, Feb 2003.

[21] Arthur Keller and Julie Basu, "A predicate-based caching scheme for client-server database architectures," in *Proceedings of the IEEE International Conference on Parallel and Distributed Information Systems*, Austin, Texas, Sept. 1994, pp. 229–238.

[22] A. Gupta, H.V. Jagadish, and I.S. Mumick, "Data integration using self-maintainable views," Technical memorandum, AT&T Bell Laboratories, Nov. 1994.

[23] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, "View maintenance in a warehousing environment," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 1995.

[24] Alon Y. Halevy, "Theory of answering queries using views," *SIGMOD Record*, vol. 29, no. 4, Dec. 2000.

[25] Jeffrey D. Ullman, "Information integration using logical views," in *Proceedings of the International Conference on Database Theory*, 1997.

[26] D. E. Knuth, *The Art of Computer Programming, volume 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1973.

[27] Donald R. Morrison, "PATRICIA—practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM (JACM)*, vol. 15, no. 4, pp. 514–534, Oct. 1968.

[28] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman Company, Nov. 1990.

[29] Theodore Johnson and Padmashree Krishna, "Lazy updates for distributed search structure," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Washington, D.C., May 1993, pp. 337–346.

[30] Witold Litwin, Marie-Anna Neimat, and Donovan A. Schneider, "LH*–a scalable, distributed data structure," *ACM Transactions on Database Systems*, vol. 21, no. 4, pp. 480–525, Dec. 1996.

[31] Miguel Castro, Manuel Costa, and Antony Rowstron, "Performance and dependability of structured peer-to-peer overlays," Tech. Rep. MSR-TR-2003-94, Microsoft Research, Cambridge, UK, December 2003.

[32] I.S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Indust. Appl. Math*, vol. 8, pp. 300–304, 1960.

[33] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. Spielman, "Efficient erasure correcting codes," *IEEE Transactions on Information Theory*, vol. 47, pp. 569–584, February 2001.

[34] Michael Luby, "LT codes," in *Proceedings of* 43th *Annual IEEE Symposium on Foundations of Computer Science*, 2002.