Query Result Caching in Peer-to-Peer Networks

Vijay Gopalakrishnan, Bobby Bhattacharjee, Peter Keleher
Department of Computer Science
University of Maryland, College Park
{*gvijay, bobby, keleher*}*@cs.umd.edu*

Sudarshan Chawathe
Department of Computer Science
University of Maine, Orono
chaw@cs.umaine.edu

Abstract

We present methods for efficient and exact search (keyword and full-text) in distributed namespaces. We extend existing techniques for single attribute queries to efficiently compute multi-attribute queries. We describe how to efficiently create, locate and store indices for implementing distributed multi-attribute searches. Our methods can be used in conjunction with existing distributed lookup schemes such as Distributed Hash Tables and distributed directories.

Our methods use a novel 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, which show that the techniques introduced in this paper can reduce search overhead (both network and processing costs) by more than an order of magnitude.

KEYWORDS

Peer-to-Peer systems, Searching, Caching, Data structures

TECHNICAL AREAS

Peer-to-Peer Systems, Data Management, Operating Systems and Middleware

I. INTRODUCTION

This paper addresses the problem of multi-attribute search in peer-to-peer (P2P) networks. We use indices of searchable attributes to permit such searches without flooding the network. Existing techniques use distributed attribute indices to allow for evaluation of single-attribute queries (e.g., `artist=x`) without flooding. In this paper, we present techniques to cache and locate query results of multi-attribute queries (e.g., `artist=x and album=y`), to improve efficiency of answering multi-attribute queries. Our work is equally applicable to systems built of either distributed hash trees (DHTs) (e.g., Chord [1], CAN [2], and Pastry [3]), or systems built from hierarchical directory services (e.g., TerraDir [4]).

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. Further, related objects (even with only slightly different names) are randomly distributed throughout the system. This random distribution, although a strength of

the DHT approach, destroys object locality. A set of objects, related by common attributes or characteristics, has its members mapped to peers throughout the system. This distribution, together with the sheer scale of the system, makes it impractical to consider solutions that (even periodically) flood the network in order to evaluate queries.

Attribute indices, which maintain a list of objects sharing a common property, emerge as an ideal candidate to address this problem. Such indices can be created and maintained incrementally. Further, the indices can be as distributed and as decentralized, just like the DHT-based applications. These indices can be stored by using the standard DHT APIs that maps each index to a peer using a hash of the index'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.

While distributed attribute indices allow for evaluation of single-attribute queries without flooding, a straightforward use of distributed indices to answer queries on two or more attributes is quite expensive. The evaluation of conjunctive queries with two or more attributes typically requires the entire index for one attribute to be sent across the network. Unfortunately, indices may be extremely large Further, attributes are often skewed according to Zipf distributions [5], potentially resulting in the size of at least some indices growing linearly with the number of system objects, and hence becoming quite large.

For example, consider the specialized query "ICDCS 06". A search on Google reports 49K valid results. To evaluate this query, we would have to intersect indexes "ICDCS" and "06", each having 274K and 590M entries respectively. A smart implementation would transfer the smaller index (6.8MB of data, assuming 26 bytes per index entry) over the network. This could further be reduced to 295KB using Bloom filters [6], or even lower using Compressed Bloom filters [7]. However, if Bloom filters are used, then all the entries of the larger index (with 590M entries) would have to be scanned, leading to a high processing cost per query. The overhead, hence, either in terms of bandwidth (if Bloom filters are not used) or processing (if Bloom filters are used), especially multiplied over billions of queries, is prohibitive.

We address the shortcomings of the above approach by using *result caching*. Previous studies (e.g.,[8]) have indicated a Zipf-like distribution for queries in P2P systems such as Gnutella. Our approach is to efficiently exploit locality in Zipf-like query streams and object attributes to cache and re-use results between queries. Our work is most applicable to systems in which queries exhibit a high degree of locality. In particular, our methods maintain not only single-attribute indices on the searchable attributes (e.g., `artist=x`, `album=y`), but also conjunctive multi-attribute indices, which we call *result caches* (e.g., `artist=x and album=y`, `artist=x and album=y and title=z`). Our methods permit efficient evaluation of conjunctive queries using these result caches by using a distributed data structure called a *view tree*. Non-conjunctive queries are evaluated by converting the query into

disjunctive normal form and evaluating the conjunctive components generated.

Results presented in Section IV show that these techniques for evaluating queries can reduce the amount of data exchanged by over 90%. 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.

The main contributions of this paper are as follows:

- We present a distributed data structure (*view tree*) for organizing cached query results in a P2P system.
- We describe how to create and maintain multi-attribute indices 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 that quantify the benefits of maintaining a view tree.

We presented a preliminary version of the idea in [9]. In this paper, we focus on the performance and utility of the view tree. We analyze in detail, with large data-sets, the cost-benefit trade-offs in using the view tree, as well as the improved search algorithm described in Section III-F. We also take into account practical issues such as load-balancing, non-uniform index sizes and failure resilience and present results for view tree performance in each of these cases.

The rest of the paper is organized as follows. Section II discusses prior work. Section III presents our search algorithm and our methods for creating and maintaining the view tree. Section IV summarizes our results. We conclude in Section V.

II. Related Work

Our work builds on recent work on peer-to-peer namespaces, including DHTs [1], [2], [3], [10], [11], and wide-area directories [4]. The techniques described in this paper can be used to provide indexed search services for all of these systems. There have been several other efforts to provide a search infrastructure over peer-to-peer systems. Reynolds and Vahdat [6], Gnawali [12], and Suel et al. [13] discuss search infrastructures using distributed global inverted indexing. Global indexing schemes have the distinguishing property of maintaining an attribute index for each keyword. Each index maps a keyword to the set of documents containing that keyword, and each host in the system is responsible for all keywords that map to it.

Reynolds and Vahdat [6] investigate the use Bloom filters to efficiently compute approximate intersections for multi-attribute queries. While Bloom filters reduce the amount of data transferred over the network, they alone are not sufficient. With Bloom filters, all the entries of the index would have to be scanned, thereby incurring a high processing cost per query. We believe that result caching and the use of Bloom filters are orthogonal techniques,

and could be used together. For example, Bloom filters can be used to compute the intersection of indices after the set of useful result caches have been identified.

In eSearch [14], Tang and Dwarakadas extract the keywords of each document using vector space models [15] and index the full text in each of the indices corresponding to the keywords. The advantage of this approach is that the network traffic to compute intersections of indices is eliminated, albeit at the cost of more storage space at each node. Note that eSearch requires only its aggregate indices, and is not affected by locality in the query stream. In Odissea [13], the authors propose a two-tier architecture to do P2P full-text searches using inverted indices.

There have also been proposals to performs searches without global indices. Tang et al. [16] propose pSearch, a scheme for semantic-based searches over the CAN [2] network. pSearch extends the vector space model [15] and latent semantic indexing (LSI) [17] 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.

There have also been proposals to build semantic overlays to improve the search in Gnutella-like networks [18]. Annexstein et al. [19] argue for combining text data to speedup search queries, at the expense of more work while attaching and detaching a peer in Gnutella-like networks. The indices are kept as suffix trees.

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

III. Searching Large Namespaces

A. 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 meta-data that we model as a set of attribute-value pairs. Attributes of an object could include keywords (for keyword search), the entire set of terms in the document (for full-text search), or other domain-specific attributes. Searching for objects by querying their indexed meta-data is the main operation that we explore in this paper.

For ease of exposition, in the rest of the paper, we assume that the attributes are boolean, i.e. either an object has a named attribute or it does not. Thus, search for attribute a would return all objects that contained attribute a. 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.

Note that the protocols in this paper readily generalize to other types of attributes (including keyword search, arbitrary word search, and range queries) depending on how the individual attribute indices (described next) are computed. Our primary focus is on the distribution, maintenance, and use of indices and result caches, and not on the methods used for managing individual indices. Specifically, we are interested in identifying and locating indices that may be used to evaluate a given query, and not on the specifics of the data structures for the indices themselves. Our methods support the identification and location of B-tree indices, hash indices or R-tree indices equally well. Similarly, our methods could be generalized easily to more structured meta-data representations, such as XML.

B. Attribute Indices

Since we wish to evaluate queries without flooding the network, we need a method for associative data access. Attribute indices provide this base functionality by mapping attribute names to objects (object names) with that attribute. Attribute indices can be stored in the P2P network in a specially designated part of the namespace. For example, the indices for attributes `manufacturer` and `price` are named `/idx/manufacturer` (We use `/idx` as the reserved namespace for indices; in practice, a more application-specific name is appropriate.) and `/idx/price`, respectively. With this naming scheme, locating indices 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 $k = \text{Hash}(\texttt{/idx/price})$.

C. Result Caches

While attribute indices 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 indices. To avoid such large data transfers, our method uses cached query results, or *result caches*.

More precisely, a result cache is the materialized result of a conjunctive query. Result Caches for non-conjunctive queries are obtained from the disjunctive normal form of the queries. The idea of using cached results to enable faster query processing has been studied extensively in the database literature (e.g., [21], [22]) and is also related to the problem of answering queries using views (e.g., [23], [24]). However, as we describe below, when result caches are scattered over a P2P network instead of located at a centralized server, the tasks of result cache maintenance and query evaluation using these result caches pose additional challenges.

We now describe the task of locating result caches 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 *location problem*, consists of determining which result caches 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: database methods assume that the set of views is well known and typically small, while in a P2P environment this set is unknown and large. The second task, the *selection problem*, consists of determining a good query plan based on the available result caches (and statistics such as cardinalities of result caches). Unlike prior work on this problem, we focus on methods that are efficient in a P2P environment without a central repository of result cache meta-data.

At first glance, it may appear that the *result cache location* problem can be solved by assigning a canonical name to each result cache (so that equivalent expressions of the same result cache are unified) and using the facilities of the P2P network for locating the named result caches. For example, if we render result caches in canonical form by listing the attribute in sorted order of their names, equivalent result caches $a \wedge c \wedge b$ 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 result cache that is equivalent to a given result cache. For example, it permits us to locate a result cache $a \wedge b \wedge c$ 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). However, this idea does not help us locate the result caches that are helpful in evaluating the above query in the absence of the result cache $a \wedge b \wedge c$. For example, the query may be answered by using result caches $a \wedge b$ and $b \wedge c$ or using $a \wedge b$ and $a \wedge c$, or using $a \wedge b$ and c, and so on. The number of result caches 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 result caches.

Any method that relies on checking for the exponentially many result caches that are relevant to a query is not practical. Some of these problems could be circumvented by maintaining a centralized list of all result caches in the network. In a P2P network, this strategy would translate to one peer storing and maintaining all indices, 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 performance hotspot and a single point of failure. Further, this list would have to be kept reasonably consistent so that updates to the data can be reflected in the cached results.

D. The View Tree

Our solution to the problems of result cache location and selection is based on a distributed data structure we call the *view tree*. The view tree maintains a distributed snapshot of the set of result caches materialized in the

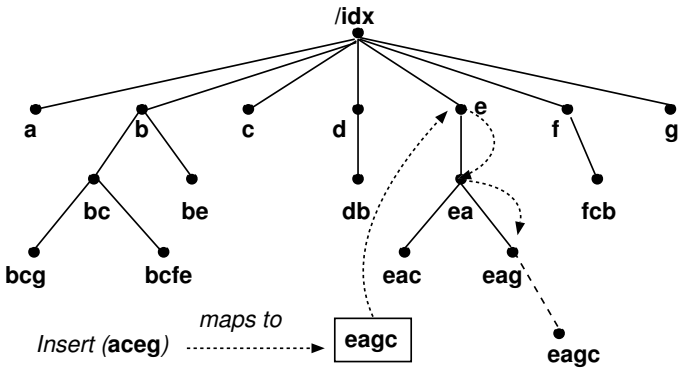

Fig. 1. Example of a View Tree. Note that a node $a_1 a_2 \ldots a_k$ represents the result cache $a_1 \wedge a_2 \wedge \ldots \wedge a_k$. The picture depicts a result cache $a \wedge c \wedge e \wedge g$ being inserted into the view tree.

network, allowing efficient location of the result caches relevant to a query. We now describe the structure of this tree, the search algorithm used to locate result caches for a query, and the methods used to maintain the tree as both data and result caches in the network change.

Each node in a view tree represents a materialized conjunctive query. 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, we refer to result caches and their corresponding view-tree nodes as simply $a_1 a_2 \ldots a_k$, the conjunctions being implicit. Each view tree node is, in general, located at a different network host. Thus, traversing a tree link potentially incurs a network hop. Figure 1 depicts an example of such a view tree. The root of the tree is labeled with the special index prefix `/idx`. The first level of the tree represents all the attribute indices (single-attribute result caches) in the network. The multi-attribute materialized results are mapped to nodes at greater depths in the tree. For example, the node bc corresponds to the result cache $b \wedge c$ and the node fcb corresponds to the result cache $f \wedge c \wedge b$.

In order to identify logically equivalent result caches (e.g., bac, cba, and bca), we refer to each result cache 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 result cache. The function is chosen so that it is equally likely to generate any one of the $l!$ permutations of a result cache with l attributes. Methods for generating such permutations are well-known [25]. All nodes use the same function; given the set of attributes, all nodes generate the same permutation.

View tree construction is illustrated in Figure 1. Suppose we wish to insert the result cache $aceg$ into the view tree. Further, suppose that the permuting function applied to this result cache results in $eagc$. The parent of this

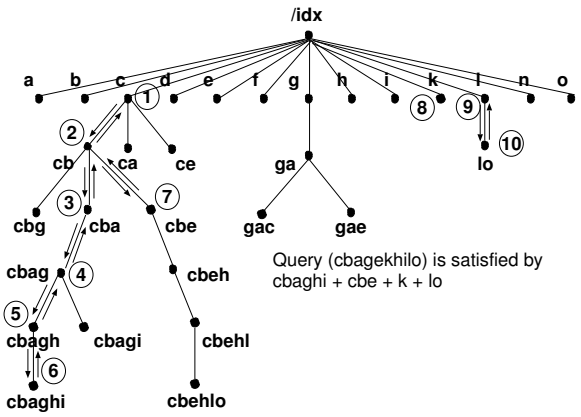

Fig. 2. Example of a search using the View Tree. The search proceeds along the direction marked with arrows visiting the nodes in order they are numbered.

result cache 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 result cache when it is inserted into the view tree is simple: we follow tree links corresponding to each attribute in the result cache, in order; the node that does not have a link to the required attribute is the parent of the node representing the new result cache.

A result cache v is deleted 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.

The policy decisions of which results 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.

E. Answering Queries using the View Tree

Given a view tree and a conjunctive query over the attributes, finding the smallest set of result caches to evaluate the query is NP-hard even in the centralized case (by reduction from *Exact Set Cover* [26]). Thus, our approach is based on heuristics. We do a depth-first traversal of the view tree such that each visited node contains at least one of the query's attributes that has not yet been covered.

For example, Figure 2 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. 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. Note that after 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 result caches $cbaghi, cbe, k$ and lo.

Each step of this search algorithm has the useful property of being guaranteed to make progress. If there is no result cache equivalent to the query, then each visited view tree node results in locating a result cache that contains at least one new query attribute (one that has not occurred in the result caches located so far).

Note that a given result cache can be used not only to satisfy a query equivalent to the one that created it, but also queries with a attribute sets that are supersets of the initial query.

F. Generalized View Tree Search Algorithm

The search algorithm described so far essentially finds a cover for the set of attributes in the query using the sets of attributes occurring in the result caches. We may define an *optimal query plan* as one that uses result caches that contain as few tuples as possible, resulting in the smallest data exchange. Given the computational complexity 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 plans that fare particularly badly by this metric. In the absence of global data statistics, we estimate the size of a result cache using the number of attributes in the result cache definition. Since our queries are conjunctive, result caches with more attributes are expected to be smaller, and thus preferred in query plans. A generalized version of our search algorithm is one that tries to find a query plan in which each result cache 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. The generalized search method stops when either all attributes in the query are covered with result caches of length (number of attributes) at least t or when none of the nodes explored so far has a child that represents a result cache of length l that can be used to replace a result cache of length $l' < t$ in the current plan. Note that the earlier search algorithm is a special case of the generalized algorithm with $t = 1$.

Figure 3 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 result caches of at least six attributes each. The search proceeds from cbe to $cbeh$ (even though $cbeh$ is not in the canonical order of the query) because the latter covers more attributes. In this process, it also finds $cbehlo$. However, it now starts searching for a six-attribute result cache that contains k. It does so by visiting every attribute index not yet visited. Since there are no six-attribute indices containing k, the search visits nodes for all 10 attributes before deciding on using $cbaghi, cbehlo$ and k to evaluate the query.

This generalization presents a trade-off between the number of view tree nodes visited and the number of tuples transmitted.

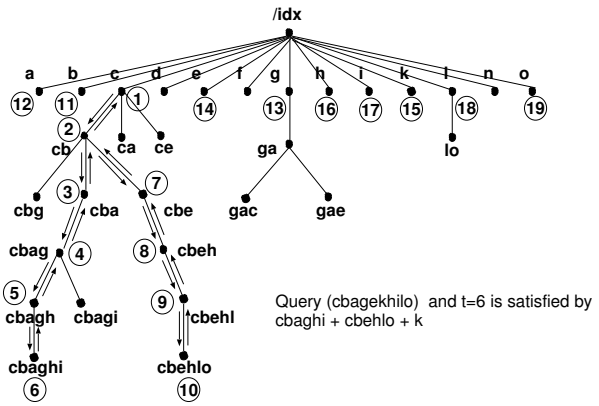

Fig. 3. Example of the generalized search algorithm. The search still proceeds along the direction marked with arrows visiting the nodes in the order they are numbered.

G. Data Modifications

Since the view tree stores result caches of network data, there is an implicit consistency requirement that the contents of a result cache be identical to the result of evaluating the corresponding query directly on the network data. This implies that insertion of an object requires that result caches containing one or more of its attributes 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 indices and result caches for several hours is not a serious problem for typical text-search applications. Further, not all indices and result caches need be updated at once. Since shorter result caches (fewer attributes) are likely to be used by a greater number of queries, the update procedure may prefer to update them sooner than longer result caches. Therefore, when an object's insertion is to be reflected in the view tree, we update result caches in order of increasing length: first, all single-attribute result caches (i.e., the attribute indices) over the object's attributes; next, all two-attribute result caches over the attributes; next, all three-attribute result caches; and so on.

The procedure for updating indices and result caches 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.

H. Handling Load Imbalances

Locality in the query stream has a downside; nodes holding indices of popular attributes will get overloaded. Our system uses the LAR adaptive replication protocol [27] to prevent the overloading of peers hosting popular indices.

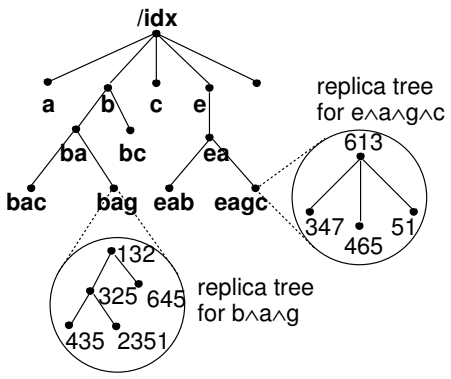

Fig. 4. Examples of replica trees.

Every peer that holds an attribute index specifies a local capacity and may request the creation of replicas when its load exceeds a threshold. By capacity, we mean the number of queries that can be routed or handled per second, and the queue length specifies the number of queries that can be buffered until additional capacity is available. Any arriving traffic that cannot be either processed or queued by a peer is dropped.

Each peer also defines a high-load threshold l^{hi} and a low load-balancing threshold l^{low}. The threshold l^{hi} indicates that a peer is approaching capacity and should shed load to prevent itself from going beyond maximum capacity and drop requests. The purpose of l^{low} is to balance the load of two peers. When the difference in the loads of two peers is greater than the low threshold, it indicates that one of the peers is less loaded compared to the other. It implies that this lightly-loaded node can be used to distribute the load. The load-based replication mechanism alleviates overloads by creating replicas hosted by overloaded peers onto lightly loaded peers.

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 at least 50% lower load. Hosts with replicas may create further replicas, resulting in a *tree* of replicas for each index. Figure 4 shows the replica trees for result caches bag and $eagc$. The numbers in the replica tree represent the server IDs where the result cache 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 peers with IDs 613 and 132 created the first result cache 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 indices have been added. On receiving this acknowledgment, the requesting peer adds a pointer to the responding peer to its list of replicas. This pointer 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.

```
1: procedure addObjectToPartition(d, t_d)
2: t_d ← t_d ∪ {d}
3: while Σ_{t∈T}|t| > S do
4:     t ← largest local index
5:     t.x = most popular b-bit prefix of t
6:     m ← a peer with sufficient space
7:     m.T ← m.T ∪ t.x
8: end while
```

Fig. 5. Node-splitting procedure. S is a per-node space limit, T is the set of all indices at a node, and b is a system parameter that dictates how large a portion of the name space is to be migrated.

Note that replication works orthogonal to the search system. As far as the view tree is concerned, all the replicas of the result cache are equivalent and can therefore be considered as a single node. Modifications to an 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 III-D; instead, they are left in place because they are reachable from the replicas.

I. Index Partitioning

In a large system with many objects, some attribute indices may be too large to be stored at any single node. Hence large indices need to be partitioned over multiple nodes. The problem of data partitioning in P2P systems has been studied previously by Ganesan et al. [28] and in eSearch [14]. Ganesan presents a scheme that uses global information to guarantee load balance among participating nodes. eSearch, which is built over Chord, maps indices to regions instead of a single point on a DHT. eSearch also modifies Chord's join protocol such that a node n joining the system contacts a set of nodes. n then identifies the most loaded node m and splits the key range that m is responsible for. To get a good distribution of keywords to nodes, existing nodes in the system have to continually re-join and remap their ranges. This leads to extra costs for transferring tuples due to re-mapping. Further, since the node IDs are no longer random, some of the underlying routing guarantees are violated (though it can be shown that the $O(\log n)$ routing guarantee in Chord is preserved, albeit with a larger constant).

We present a simple partitioning scheme that adaptively distributes large indices over multiple nodes. Our design of the partitioning scheme is motivated by three guidelines: (1) Intersections of sets of object identifiers (required for answering conjunctive queries) should be efficient, (2) The overhead of updating indices in response to insertion of new objects and other changes should be small, and (3) The partitioning scheme should not depend strongly on the underlying P2P object-location architecture. Our partitioning scheme is summarized by the pseudo-code in Figure 5. When an object d appears in the network (perhaps as a result of a node joining the network), the remote procedure *addObjectToPartition* is invoked for d and the attribute indices t_d for each of d's attributes. The procedure

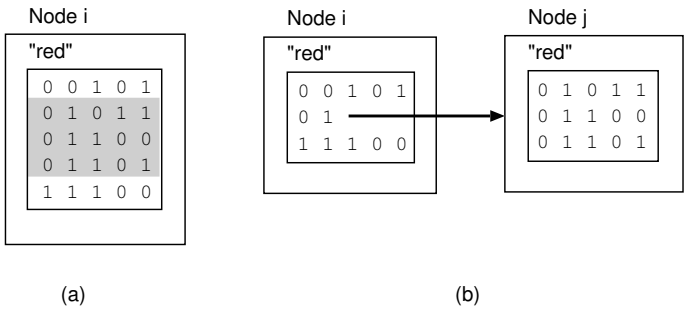

Fig. 6. Node Splitting: In (a), node i has a single index for the attribute "red", containing the names (in binary) of five red objects. In (b), node i has split the index by sending the popular 2-bit prefix "0 1" to node j. Nodes i and j form a (quite short) partition tree.

is invoked at the node known to be the root of the index t_d as identified by the mechanisms of the underlying P2P framework (for example, by looking up t_d in a DHT).

We assume that each participating node allocates some fixed amount of disk space, say S, for storing indices. Partitioning is invoked when the storage exceeds this threshold (line 4). In this case, the largest index (t, line 5) is chosen for partitioning. The index is partitioned by dividing its object identifiers into equivalence classes based on common b-bit prefixes, where b is a parameter. The largest such partition ($t.x$, line 6) is migrated to another peer. This peer is chosen by randomly, by sampling, until one with sufficient space is found.

Figure 6 shows an example of this algorithm. We set $b = 2$ for the example. In the figure, the prefix "01", is the largest partition and is moved to another peer j. Repeated application of the partitioning scheme outlined above results in the formation of a tree of partitions for each index, rooted at the node that is the original host for the index. We refer to this tree as the *partition tree*. For e.g. in 6, if node j needs to split, it will use bits 3 and 4 to partition the index further, creating a partition tree. Index lookups proceed by locating the partition tree's root using standard P2P services and proceeding down the tree as necessary. It is important to keep in mind that the partition tree, like the replica tree, is orthogonal to the view tree and is over an individual result cache in the view tree.

J. Failure Recovery

If the index for an attribute were to become unavailable due to node or network failures, the performance of queries containing the attribute would suffer. 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 to generate complete results. Further, since a single large index can span multiple nodes due to partitioning, the probability that a large index is available depends on *all* of its host nodes being available. Thus, indices (especially those that require flooding to recreate) must be redundantly stored. The level of redundancy depends on the probability of simultaneous node failures. Once the requisite number of static replicas are created, we can use standard techniques, such as those based on

heartbeats [29], to maintain the proper number of replicas. Note that the locations of the current set of static replicas for a index, including all partitions, must be consistently maintained at all other replicas.

The performance of update operations depends on the method used to make indices redundant. The most obvious (and often optimal) approach is to keep a literal copy of the index data at each replica. Note that these static replicas are only used for failure recovery and not for answering queries: we employ separate load-based replicas created using LAR for that purpose. It is also possible to use erasure coding, e.g. Reed-Solomon codes [30], to add redundancy into the system. In order to guarantee the same levels of resilience, Erasure codes require lesser disk space compared to replication. The reduced space, however, comes at an increased update cost because the entire set of erasure codes must be re-published for each update.

Assume that the independent node failure probability is f. Further, assume that each index is partitioned into m partitions. Then, if each partition is replicated R times, then the probability of not answering a query is given by

$$1 - \left(1 - (f)^R\right)^m \tag{1}$$

On the other hand, if each partition is coded into n blocks, of which any k is sufficient to re-construct the partition, then the probability of not answering a query can be computed using Chernoff bounds and is at most

$$1 - \left[1 - e^{-n \cdot (1-f)}\right]^m \tag{2}$$

From equation 2, we note that if the nodes in the system fail with independent probability f, and we want to recreate an index (split into m parts) with probability $1 - p$, with simple replication, we would require $R = \frac{\ln(1-(1-p)^{1/m})}{\ln(f)}$ copies for each partition, which leads to $m \cdot \lceil R \rceil$ copies in total for the entire index. Instead, if we assume that using erasure codes, any k surviving nodes can recreate a partition, then for each partition, we only require $\frac{-\ln(1-(1-p)^{1/m})}{(1-f)}$ nodes for the same probability of recovery. For $m = 32$, $f = .1$, $k = 8$, and $p = 10^{-4}$, taking rounding into account, this analysis implies replication requires 3.2 times more space than erasure coding.

While coding reduces the amount of storage required, it increases the cost of processing updates, because when an index is updated, we have to reassemble the entire index (potentially visiting a number of nodes), apply the update, and create new encoded blocks. This procedure can be made more efficient by encoding fixed ranges of the index into independently coded blocks such that only the affected ranges have to be reassembled upon updates. Finally, the cost of updates to encoded indices can also be amortized by applying updates in batch. We explore the space-update overhead of encoded index storage in Section IV-F.

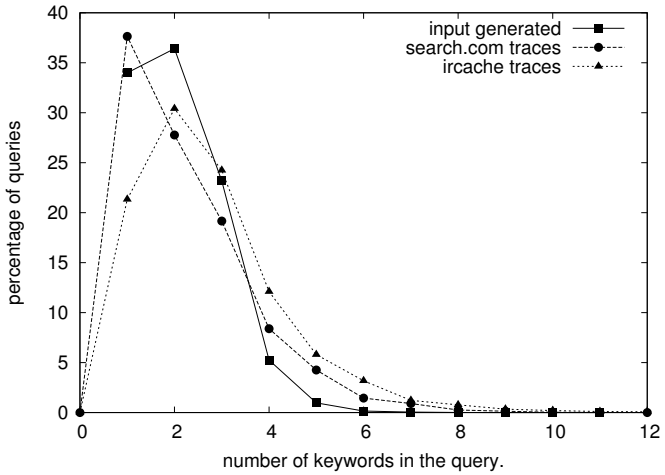

Fig. 7. The distribution of keywords in queries in our generated input and in traces obtained from search.com and IRCache.

IV. RESULTS

This section presents simulation results from our experiments with the algorithms described in section III. We implemented a packet-level simulator for evaluating our algorithms. We chose documents from the WT-10g data set of the Text REtrieval Conference (TREC [31]) data as our source data. We generated the mapping between data-items and keywords using the term-weighting schemes normally used in information retrieval methods like Vector Space models [15]. The term-weighting scheme computes the weight t_i of the term i in a document using equation 3, where f_i is the frequency of the word i in the document, n_i is the total number of documents that have the term i and N is the total number of documents in the collection.

$$t_i = (\ln(f_i) + 1) \cdot \ln\left(\frac{N}{n_i}\right) \tag{3}$$

The weights of terms in each document are then normalized and terms with weights greater than a threshold are considered as keywords for the document. For our data, we experimented with different values of threshold and settled on 0.12 as it gave a good distribution of keywords and index sizes. This resulted in a collection of 500k documents with 390k unique keywords. The number of keywords per document ranged from 1 to 55 with an average of 14.

For the queries we first chose a representative sample of Web queries from the publicly available `search.com` query set and from the cache logs available from `ircache.net`. These web query sets do not provide an associated document set over which these queries would be valid. Hence, we could not directly use these queries for our experiments. We, therefore, generated queries with the same characteristics (distribution of attributes over queries, number of attributes per query, etc.) as the trace queries using keywords from the TREC-Web data set. Figure 7

# of attributes	90% locality			10% locality		
	min	avg	max	min	avg	max
1	1	18	8514	1	18	8514
2	0	39	6540	0	6	6540
3	0	39	4776	0	6	4776
4	0	17	4480	0	2	3887
5	0	3	3614	0	1	3614

TABLE I

SIZES (IN NO. OF TUPLES) OF n-ATTRIBUTE INDUCER.

shows the distribution of keywords in our queries. We generated query streams consisting of 1,000,000 queries. In order to represent locality in the query stream, we generated a *popular set* of 100,000 queries (10% of the total queries). Each query stream was generated by drawing either 5%, 10%, 20%, 50%, 90% or 99% of the queries, uniformly at random, from this popular set. The reminder of the queries, in each of the query streams, were generated by determining the size from the distribution and picking individual keywords uniformly at random from the entire set of keywords. By default, we used the query streams with 90% and 10% of queries from the popular set as the set with high- and low-locality respectively. Table I shows the sizes of resulting attribute indices and cached results depending on the locality in the query stream. Note that our query stream generated a small number of indices with more than five attributes, but their effect is negligible and they are not show here.

A. Experimental setup

The base system for our experiments consisted of 10,000 servers exporting 500,000 documents. By default, we ran each experiment with the 90% locality query stream. The query inter-arrival time was exponentially distributed with an average of 10 milliseconds. We assumed that hosts allocate some fixed amount of disk space to store the result caches. In most of our experiments, we *did not* take into account the space taken up by the attribute indices. We did this both for correctness of the results and because we were interested primarily in understanding how various factors affected the performance of view trees. We allocated 750 tuples of space at each node by default, for the multi-attribute caches. This space was managed as a cache, with cached results being deleted from the view tree when marked for replacement by LRU. Unless otherwise noted, we set the search parameter $t = 1$ in the generalized search algorithm; this results in the use of the regular view tree search algorithm. The rest of this section discusses the detailed experiments we performed and analyzes the results.

B. Effectiveness of Result Caching

Our first experiment quantifies the effectiveness of result caching. We consider six different input query distributions, each with varying degrees of locality. We start with a query stream with 5% of queries (50000) from the 100k working set and gradually increase locality till 99% of the queries are from the working set. We plot the

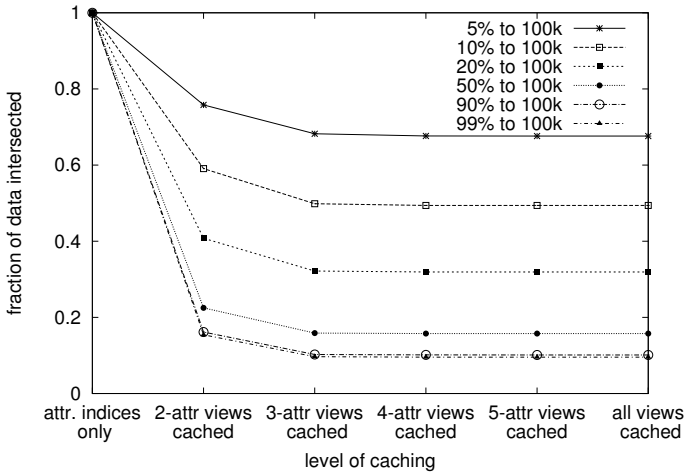

Fig. 8. Caching benefit by level.

result of the experiment in Figure 8. The x-axis of the represents the level of caching in the view tree. There are *no* caches when we have only attributes indices. We increase the amount of caching in each step by progressively caching 2-attribute result caches, 3-attribute result caches and so on until we cache all result caches. The y-axis presents the fraction of data exchanged for computing the results. We *normalize* the amount of data exchanged for each caching level with the case when there are no caches.

Figure 8 shows that result caching significantly reduces the amount of data transferred for query evaluation. The benefit is enormous for streams with high locality; when 90% of the queries are from the working set, caching 2-attribute indices reduces the number of tuples transferred by 85%, whereas caching all indices reduces the number of tuples transferred by over 90%. Somewhat surprisingly, queries with much lower locality(5% and 10%) also benefit from result caching. This is because the intersection of two random keywords usually results in a small index, which can be efficiently cached and reused.

Finally, note that Figure 8 shows that a majority of the view tree's benefit coming from only the first two or three levels of the trees. This is because our query stream consists primarily of queries with few attributes (see Figure 7).

In Figure 9, we plot, over time, the number of hits to different k-attribute result caches. The plot also shows the actual number of queries with different numbers of attributes over time. For each curve, we sum the hits and the queries over a 30-second period.

There are several interesting points to note: first, result caching is effective, in that the number of (costly) accesses to the single attribute indices decrease rapidly as the multi-attribute caches are built. Second, after the two attribute indices are built, almost all two attribute queries are satisfied using these caches. Ideally, the curve for hits

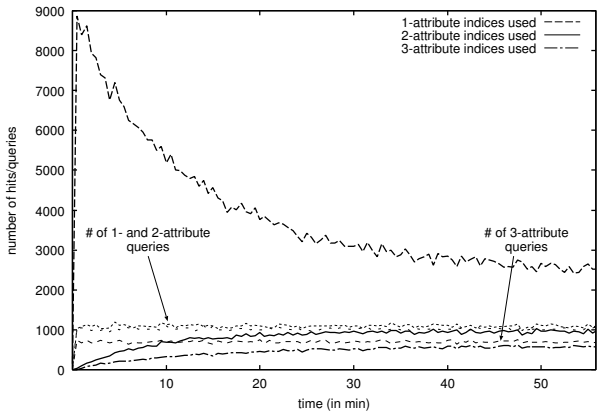

Fig. 9. Index access and k-attribute queries over time.

	10% locality		90% locality	
# of Updates	# of Extra Updates	Reduction in data transferred	# of Extra Updates	Reduction in data transferred
1k	344	95.50 M	654	165.99 M
10k	2064	95.53 M	5628	165.99 M
100k	24113	95.71 M	57989	166.00 M
1M	210k	97.33 M	534k	166.08 M

TABLE II

UPDATE OVERHEAD FOR 1,000,000 QUERIES. ALL THE RESULTS ARE IN NUMBER OF TUPLES.

on single-attribute indices would converge with the line for single-attribute queries, and likewise for indices and queries with more attributes. They do not because of the multiplicity of the 3- and higher-attribute queries: there are so many distinct sets of attributes that it is not feasible to cache them all (or for the intermediate result caches to cover them all), and satisfying these queries sometimes requires using single-attribute indices.

To evaluate whether caching larger result caches is more useful for queries with more attributes, we looked specifically at the data for these queries. The results show that the majority of result caches used in satisfying queries of more than two attributes have more than two attributes, i.e., deep view trees can be quite effective for queries with many attributes. In fact, the majority of all matches in the view tree turned out to be exact matches. One explanation is that this is, to some extent, a function of a query stream in which the "popular set" queries repeat. We tested this theory by running another set of experiments where the occurrence of individual attributes had locality, instead of specific multi-attribute queries. As expected, we found a much lower rate of exact matches and a higher number of useful hits where the view is a subset of the query.

C. View Maintenance: Updates

Table II summarizes the cost and benefit of maintaining the view tree. In these experiments, we assume that we have infinite disk space and cache *all* result caches (which corresponds to the worst case for update overheads) and

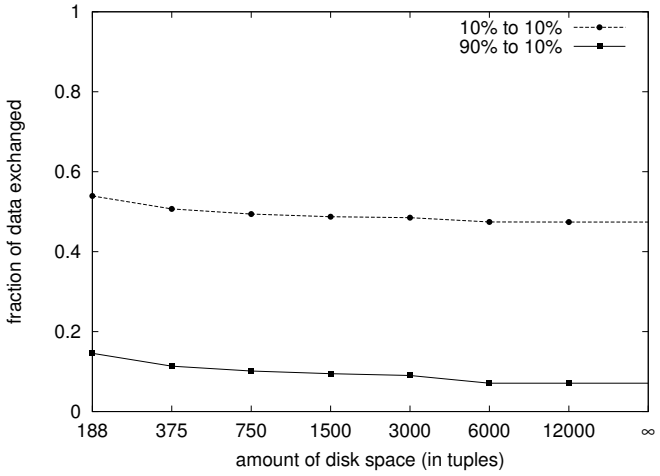

Fig. 10. Effect of disk space: Even a small fraction of space allocated for the caches helps reduce the number of tuples intersected.

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.

In Table II, we present (1) the number of *extra* updates needed 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 cost of maintaining the view tree is essentially negligible.

The number of updates is higher in the query stream with higher locality. This is because each update has to propagate to more caches. 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.

D. Effect of Disk Space

This experiment quantifies the effect of the amount of disk space on the efficacy of the view tree. Recall that in our simulations, the nominal amount of per-node disk space is 750 tuples. In Figure 10, we plot the performance of the view tree for disk space allocations ranging from 188 tuples (0.25×750) to 12000 tuples (16×750). The figure also contains a result with unbounded disk space (∞). We assume that keeping any information takes up space; caches with zero entries also take up space.

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). The interesting aspect, however, is that disk space does not affect the performance of the view tree. For example, with 90% locality in the query stream, the number of tuples exchanged drops by 83% with only 188 tuples. This surprising result can be explained by the fact that most of the multi-

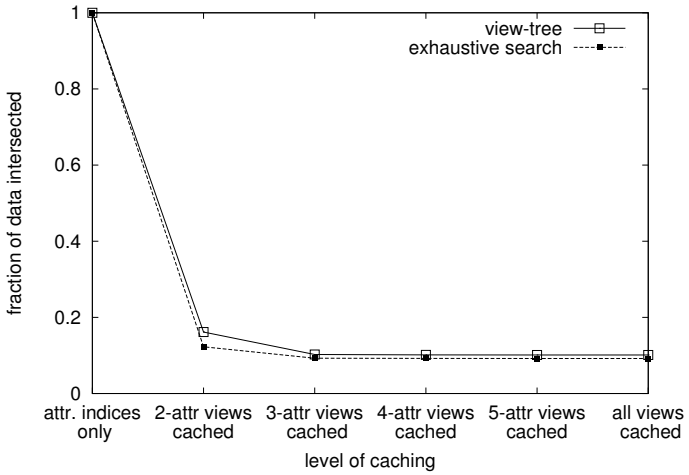

Fig. 11. View Tree Organization (90% working set).

attribute indices are small (refer to Table I) and hence can be easily stored. Also, for our experiments, at 12000 tuples, the performance is almost equivalent to having unbounded space.

E. Tree Organization and Search Procedure

Recall that the view tree search algorithm (Section III-F) is based on heuristics, and is not guaranteed to find all useful indices. Figure 11 compares the bandwidth reductions achieved by the view tree heuristic with the best possible reduction, as discovered via exhaustive search. We computed the results for exhaustive search using result caches that result in the smallest data exchange, just as with the unmodified view tree algorithm.

The results show that both organizations perform similarly; the view tree approach differs from the exhaustive search by a maximum of 5% when we cache two-attribute indices. As we increase the level of caching, the difference in performance stays around 1%.

F. Encoded Storage vs. Replication

Recall that static replicas and erasure codes are alternate strategies for providing resilience. Here, we analyze the trade-offs while using each of the strategies. For this experiment, we fix the number of partitions of the index to 32 and assume that the size of each partition is 100 times the size of an update (i.e., each partition holds 100 tuples, and an update changes a single tuple). We assume that any eight erasure-coded fragments are required to re-create the partition.

Table III compares the amount of storage required by erasure codes and replication to re-create the data when the nodes in the system fail with different (independent) failure probabilities. We describe the costs as the *ratio* of the costs due to replication to the cost using erasure codes. Thus a value above 1 indicates that erasure codes are better, while values less than 1 indicate that replication is better.

Failure prob.	Index Size (EC = 1)	Update Cost (EC = 1)	
		Online	Batch
0.01	1.85	0.018	0.46
0.10	3.20	0.032	0.80
0.20	4.00	0.040	1.00
0.50	5.85	0.058	1.46
0.90	7.62	0.076	1.91

TABLE III

NORMALIZED SIZE AND UPDATE COST FOR REPLICATED INDICES FOR DIFFERENT NODE FAILURE PROBABILITIES.

In each row of the table, we choose an independent probability with which each node in the system fails, and create sufficient replicas, using our analytical bounds, such that the index can be recovered, using either replication or erasure coding, with probability $(1 - 10^{-4})$. The "Index Size" column shows the space required for replication as a factor of the space required for coded indices. Replication uses 2–8 times more space depending on the failure probability. However, the trade-off is in the amount of data transferred during an update. When individual updates are immediately applied ("Online" column in Table III), pure replication saves between 93–99% of the data transfers required by encoded storage. As we pointed out earlier, however, the disparity can be significantly reduced if updates can be batched ("Batch" column in Table). Here, we batch 25 updates, and the cost for updating coded indices is reduced to at most twice that of replication, for even very low failure rates. Interestingly, for high enough failure rates, encoding can even be *more* efficient for batch updates than replication. Obviously, the particular storage scheme used by any system will depend on the expected operating regime of the deployment: our results indicate that encoded storage is preferable when (1) update rates are very low, or (2) failure rates are high, or (3) if updates can be applied as batches.

G. Handling Popular Indices

Popular indices can induce load imbalance across view tree nodes. We address this imbalance with two techniques in this paper: (1) adaptive replication via LAR, and (2) the use of permutation in defining canonical result cache names. Figure 12 plots the fraction of queries answered versus caching level both with and without adaptive replication. For the latter case, we plot data for the system with and without name permutation. In this experiment, each peer can process 10 queries/second and can buffer a maximum of 32 queries. All queries received when the buffer is full are dropped. We carefully experimented with the system load to make sure that queries were not dropped because of a high query rate but rather because of the imbalance in the tree.

Figure 12 shows that adaptive replication is quite effective in distributing load (as indicated by the fraction of queries answered). Almost all queries (>99.99% in all cases) are answered irrespective of the level of caching.

Increasing the level of caching is also effective at reducing the number of dropped queries. With the non-permuted

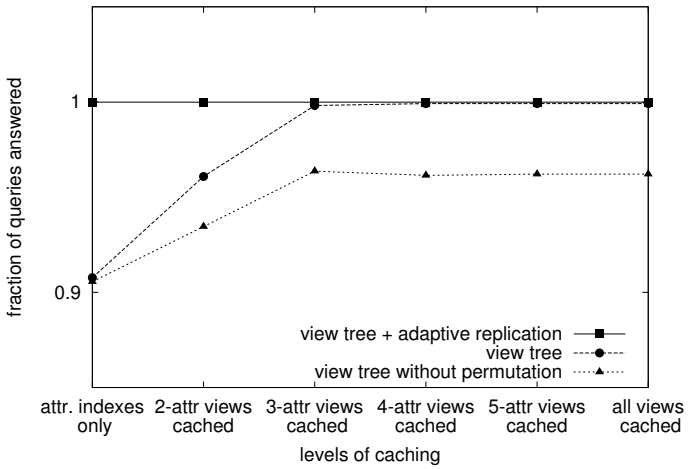

Fig. 12. Queries answered with and without adaptive replication. Note that the y-axis ranges from 0.85 to 1.05

	Attribute indices	2-attribute result caches	3-attribute result caches	4-attribute result caches	5-attribute result caches	All result caches
# of replicas created	1281	904	411	327	339	339
data transferred	2.04 M	1.56 M	1.26 M	1.05 M	1.06 M	1.06 M

TABLE IV

OVERHEAD OF USING THE ADAPTIVE REPLICATION WITH VIEW TREES.

view tree, the number of queries answered increases from 91% to 96% with increasing level of caching, while almost all of the queries are answered with three-attribute caches and the regular view tree. The increase is because more caches imply a higher hit rate and more exact matches, and both of these effects reduce bandwidth requirements. More caches also imply that the query load is distributed across a larger set of peers.

Figure 12 also shows the merit in creating the view tree by permuting the keywords identifying the index. The number of queries answered increases by an additional 5% with permutation, indicating that permuting the keywords facilitates the distribution of load over the nodes.

Adaptive replication comes with its own costs: the network overhead of creating replicas and the extra space taken up by these replicas. We quantify both costs in Table IV. The first row in Table IV counts the number of replicas created per level of the view tree, while the second row tabulates the extra network traffic due to transferring these adaptive replicas. Note that in the worst cases (with attribute indices and 2-attribute result caches), the cost of creating adaptive replicas (about 1300 create messages and 2 Megabytes of control messages over 1M queries) is negligible.

H. Handling Large Indices

In section III-I, we have argued that, in systems with many documents, indices have to be partitioned onto multiple nodes. To understand the importance of partitioning we ran two sets of experiments.

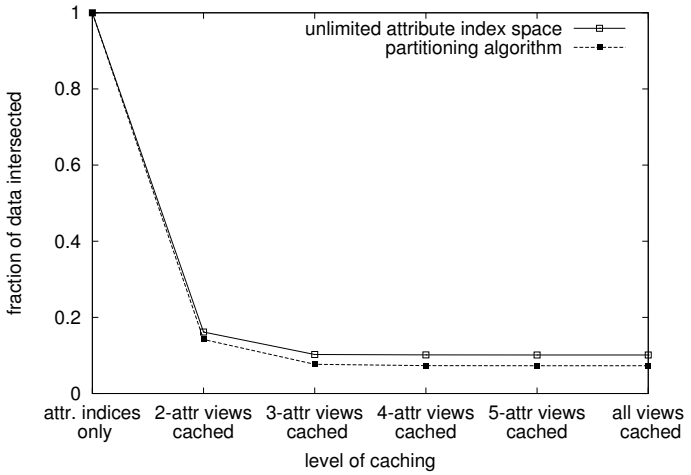

Fig. 13. Performance of view tree with partitioning

The first experiment measured the importance of partitioning to maintain correctness of the indices. To perform this experiment, we allowed each node to have 1500 tuples of disk space. This space was utilized to store both attribute indices and cached results. The average space required to store just attribute indices was 750 tuples. We allow an extra space of 750 tuples to store cached results. If a node runs out of space, it neither adds new entries in the attribute indices hosted on the node, nor does it cache new results. Note that this is different from the experiments presented so far where we allowed the entire attribute index to be stored at a node, irrespective of its size. The amount of disk space for caches, however, is consistent with all our experiments. Our experiments showed that without partitioning, *only 28% of the queries were accurately answered*. When partitioning was employed we were able to answer *all* the queries accurately for the same amount of disk space

The second experiment measured the effect of partitioning on result caching. We compared the result in Figure 8 with 90% locality to the amount of data transferred for the same query stream with partitioning employed. In the case of partitioning, we also account for the data moved by the partitioning algorithm. Figure 13 presents the results of this comparison. It is clear from the figure that partitioning helps further reduce the amount of data exchanged.

I. Synopsis of Other Results

We performed several other experiments which we only summarize . We ran experiments similar to the ones presented with different kinds of query streams and with a different document set. The document set consisted of a small set of keywords but resulted in indices that were much larger. We ran experiments with query streams that exhibited locality on the attributes rather than using a query bag. We also ran experiments with query streams that had as many multi-attribute queries as 2- and 3-attribute queries. The results are similar to those shown in this paper as long as there is locality in the query stream.

V. Conclusion

We have described the design of a 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. 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 techniques for reconstructing the tree upon failures. We discuss and present algorithms for some of the practical considerations associated with implementing the view tree including load-balancing and failure resilience,

Our results show that using a view tree offers significant benefits over maintaining simple one-level attribute indices. With our trace data, view trees reduce multi-keyword query overheads by over 90%, while consuming few 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 show that view trees efficiently enable much more sophisticated document retrieval on P2P systems.

References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. 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] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[3] A. Rowstron and P. 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.

[4] B. Bhattacharjee, P. Keleher, and B. Silaghi, "The design of TerraDir," University of Maryland, College Park, MD, Tech. Rep. CS-TR-4299, October 2001.

[5] G. K. Zipf, *Human Behavior and the Principle of Least Effort*. Cambridge, Massachusetts: Addison-Wesley, 1949.

[6] P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," in *Proceedings of IFIP/ACM Middleware 2003*, 2003.

[7] M. Mitzenmacher, "Compressed bloom filters," in *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. ACM Press, 2001, pp. 144–150.

[8] K. Sripanidkulchai, "The popularity of gnutella queries and its implications on scalability," February 2001. [Online]. Available: http://www.cs.cmu.edu/-kunwadee

[9] B. Bhattacharjee, S. Chawathe, V. Gopalakrishnan, P. Keleher, and B. Silaghi, "Efficient peer-to-peer searches using result caching," in *Proceedings of the* 2^{nd} *International workshop on Peer-To-Peer Systems*, Berkeley, CA, March 2003.

[10] I. Gupta, K. Birman, P. Linga, A. Demers, and R. 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.

[11] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. 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.

[12] O. D. Gnawali, "A keyword set search system for peer-to-peer networks," Master's thesis, Massachusetts Institute of Technology, June 2002.

[13] 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. [Online]. Available: http://cis.poly.edu/suel/papers/odissea.pdf

[14] C. Tang and S. Dwarakadas, "Hybrid global-local indexing for efficient peer-to-peer information retrieval," in *Proceedings of USENIX NSDI '04 Conference*, San Fransisco, CA, March 2004.

[15] G. Salton, A. Wong, and C. Yang, "A vector space model for information retrieval," *Journal for the American Society for Information Retrieval*, vol. 18, no. 11, pp. 613–620, 1975.

[16] C. Tang, Z. Xu, and S. Dwarkadas, "Peer-to-peer information retrieval using self-organizing semantic overlay networks," in *Proceedings of ACM SIGCOMM '03 Conference*. Karlsruhe, Germany: ACM Press, 2003, pp. 175–186.

[17] S. Deerwester, S. Dumais, T. Landauer, G. Furnas, and R. Harshman, "Indexing by latent semantic analysis," *Journal for the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[18] E. Cohen, A. Fiat, and H. 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] J. Li, B. T. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. 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] A. Keller and J. 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] 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.

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

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

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

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

[27] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher, "Adaptive replication in peer-to-peer systems," in *The 24th International Conference on Distributed Computing Systems*, Tokyo, Japan, March 2004.

[28] P. Ganesan, M. Bawa, and H. Garcia-Molina, "Online balancing of range-partitioned data with applications to peer-to-peer systems," in *Proceedings of the* 30th *VLDB Conference*, Toronto, Canada, August 2004.

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

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

[31] "Text REtrieval Conference." [Online]. Available: http://trec.nist.gov/