Efficient Peer-To-Peer Searches Using Result-Caching

Bobby Bhattacharjee, Sudarshan Chawathe, Vijay Gopalakrishnan, Pete Keleher, Bujor Silaghi
{*bobby, chaw, gvijay, keleher, bujor*}*@cs.umd.edu.*

1 Introduction

Existing peer-to-peer systems implement a single function well: data lookup. There is now a wealth of research describing how to reliably disseminate, and to later retrieve, data in a scalable and load-balanced manner.

However, searching has received less attention. The current state of the art is to distribute inverted indexes in the name space. Intersection of distributed sets can be made more efficient by exchanging bloom filters prior to moving objects [2].

This paper proposes an orthogonal and complementary technique: using result-caching to avoid duplicating work and data movement. For example, assume that indexes a_i, a_j, and a_k are located on distinct nodes in the network. Computing $a_i \wedge a_j \wedge a_k$ directly from these indexes is much more expensive than intersecting the result of a prior $a_i \wedge a_j$ operation together with a_k.

The main contribution of the paper is a new data structure, the *view tree*, that can be used to efficiently store and retrieve such prior results. These results, which can also be thought of as materialized views, can then be used to efficiently answer future queries. Note that object attributes could either be derived from application semantics (e.g. meta-data from files in a filesystem) or computed via techniques such as latent semantic indexing.

1.1 Data and Query Model

We assume that each data item in the namespace has a unique name, and has some searchable meta-data associated with it. We assume the meta-data is represented as an ordered set of attribute-value pairs. The attributes may be boolean or may be real valued. It is possible to extend our search scheme to handle more complex meta-data schemes, including hierarchically arranged attribute trees, but we do not consider this extension in this paper.

Our queries have the form $(a_i \wedge a_j \wedge \ldots \wedge a_k) \vee (b_i \wedge b_j \wedge \ldots \wedge b_k) \vee \ldots \vee (n_i \wedge n_j \wedge \ldots \wedge n_k)$. The solution to such a query is the union of the solutions of each conjunctive clause. For example, if the query is $(a \wedge b) \vee (b \wedge c)$, where a, b, c are boolean predicates, then the result is the union of the items that have either attribute $(a \wedge b)$ or attribute $(b \wedge c)$. We use the term "view query", or just "query", to refer to such boolean queries, and the term "view" to refer to a set of namespace elements that satisfy a particular view query.

The rest of the paper is organized as follows. Section 2 presents the details of our search algorithm, together with the creation and maintenance of the view tree. Section 3 describes preliminary results, Section 4 discusses prior work, and we conclude in Section 5.

2 The View Tree

The core of our method works with conjunctive queries. Queries with disjunction are first converted to disjunctive normal form (disjunction of conjunctions), and each conjunction is evaluated as a separate conjunctive query. The results of the conjunctions are cached separately. For example, the evaluation of the query $(a \wedge b) \vee (b \wedge c)$ results in two views being cached: $(a \wedge b)$ and $(b \wedge c)$. These views can subsequently be used to answer the original query, and also other queries that contain these views. Henceforth, we shall discuss only conjunctive queries.

The views corresponding to conjunctive queries can be located by searching for the view using a canonical representation. In a distributed hash tree (DHT), the views are stored at nodes where the hash of the canonical name maps to. For example, in Chord, the view $a \wedge b$ is stored at the successor of $H(\text{“}a \wedge b\text{”})$. Note that the same technique is used to find the nodes where each attribute index should be stored. In a hierarchical system, the views can be stored in a hidden part of the name tree with each view stored at the server that initially creates the view. Obviously, more sophisticated techniques that balance the storage load can also be used in a hierarchical system. In both cases, views are located by searching the namespace using the canonical representation.

Unfortunately, merely storing each view in the namespace with a canonical name is not sufficient to efficiently answer view queries, even if the underlying namespace can very efficiently locate each materialized view. For a single conjunctive query $a_1 \wedge a_2 \wedge \ldots \wedge a_k$ with k attributes, the number of views that are useful for evaluating the query is exponential in k. Clearly, for moderately large k, it is not feasible to search the namespace to for each of these views to determine which ones exist. One solution is to maintain a central, consistent list of currently materialized views. Useful views that are materialized can then be obtained by locating this list in the namespace. The problems with a naive central list are many and obvious. For example, it has to be updated whenever any new view is created or destroyed in the system, which can happen whenever a new query is satisfied or when a node storing a view fails. Further, the storage required at the node holding the list is of the order of the number of views in the system.

Therefore, we propose the *view tree* for maintaining a distributed snapshot of the set of currently materialized views. The view tree can be traversed to find all relevant views for a particular query. The view tree can easily be implemented as a trie and, as we discuss later, does not have the state and update scalability problems of a central list solution. An example of a view tree is depicted in Figure 2. Nodes in the view tree are labeled with attributes. To locate the node at which a view $a_1 a_2 \ldots a_k$ is stored, we descend from the root of the view tree, in standard trie manner, first to the child labeled a_1, then to its child labeled a_2, and so on. If this process stops before we reach the end of the string representing the view, the view is stored as a child of the last node reached. Thus, the view tree is a trie in which each node that has no siblings is merged with its parent.

Since we wish to always provide efficient attribute-based access to objects, we require that all single-attribute views be materialized. There are therefore as many nodes at depth one in the view tree as there are attributes in our domain, and each node stores the view for the corresponding attribute. Thus, the method for adding a node to the underlying namespace is augmented to also update the index for each attribute that is part of the new node's meta-data. Similarly, these attribute indexes are also updated as nodes are deleted from the namespace.

We define a canonical order on the attributes and use this order to uniquely identify equivalent conjunctive queries (and views). For example, assuming an alphabetic ordering of attributes, $a \wedge b$ and $b \wedge a$ map to the canonical form $a \wedge b$ (or simply ab in our abbreviated notation). Henceforth, we will assume that all queries and views are in canonical form. Since the canonicalization is performed by the system, this assumption is without loss of generality and is transparent to the user of our system.

2.1 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 [10]). Thus, an exact solution is not practical, especially in a distributed setting. Instead, we use the following guidelines for a method for locating views for answering a query: (1) *exact match* - If a view that matches the query exactly exists, that view must be located. (2) *forward progress* - If there is no exact match, then each view tree node that is visited must result in locating a view that contains at least one attribute from the query that does not occur in the views located so far.

```
1:  Let $\{n' : p(n') = n\} = \{c_1, \ldots, c_n\}$
2:  $q_w \leftarrow q$
3:  $r_w \leftarrow \emptyset$
4:  loop
5:    if $q_w = \epsilon$ then
6:      return $r_w$
7:    else
8:      Let $q_w = a_1 \ldots a_m$
9:      if $\nexists i, j, s_1, s_2 : s_1 \| a_i \| s_2 = r(c_j)$ then
10:       return $r_w$
11:     else
12:       $(i^*, j^*) \leftarrow \min\{(i, j) : \exists s_1, s_2 : s_1 \| a_i \| s_2 = r(c_j)\}$
13:       $r_w \leftarrow r_w \cup \{r(c_{j^*})\}$
14:       $r' \leftarrow S(c_{j^*}, a_{i^*+1} \ldots a_m)$
15:       $q_w \leftarrow q_w \ominus r'$
16:       $r_w \leftarrow r_w \cup r'$
17:     end if
18:   end if
19: end loop
```

Figure 1: Search for query q at node n: $S(n, q)$

Our search algorithm is outlined in Figure 1. We use $p(n)$ to denote the parent of a node n, $l(n)$ to denote the string formed by the concatenation of attributes indexed at n, and $r(n)$ to denote the suffix of $l(n)$ such that $l(n) = l(p(n)) \| r(n)$. We use the notation $s \ominus X$ to denote the string obtained by deleting from s the characters that occur in set X. Given a query q, the algorithm is invoked as $S(r, q)$, where r is the root of the view tree. The computation at a node n (i.e., $S(n, q)$) 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 impracticable. It is easy to verify that the test on line 12 ensures that both the conditions in our guidelines (exact match and forward progress) are satisfied. 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.

For example, Figure 2 shows a search for "cbagekhilo", which proceeds as follows. The algorithm first locates the best prefix match, which is "cbag" in this case. Even though the next clause in the prefix, "cbage" has not been 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 add more unresolved literals.

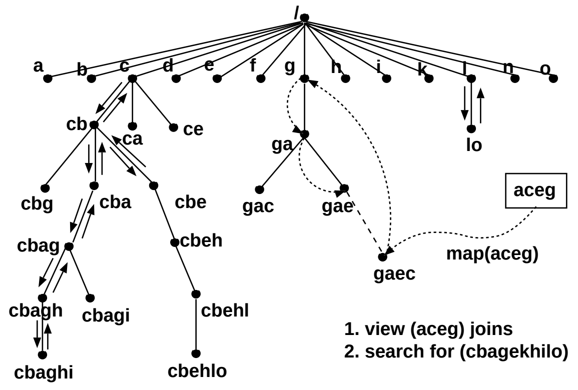

Figure 2: Example of a node joining the View Tree.

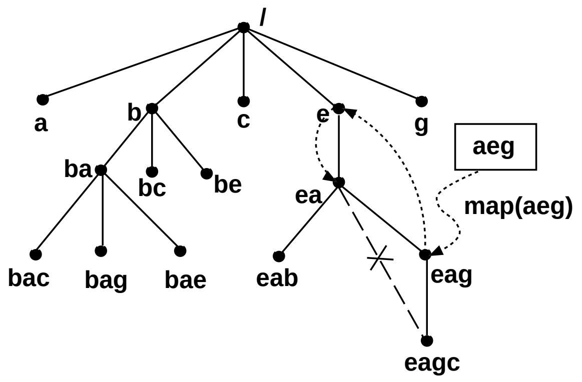

Figure 3: Maintaining the tree when a new node joins.

2.2 Creating a Balanced View Tree

In our description, we have implicitly assumed that a view of the form $(a \wedge b \wedge \ldots \wedge x)$ can only be part of the subtree starting at a, since a occurs first in the canonical representation of $(a \wedge b \wedge \ldots \wedge x)$. This will result in well-defined trees, but they will be heavily imbalanced with most of the views stored under the attributes which occur first in the canonical representation. For example, all views of the form $a \wedge \ldots \wedge x$ will be stored under the a subtree. Obviously, a view of the form $(a \wedge b \wedge \ldots \wedge x)$ can equally well be stored under a clause of the form $(b \wedge \ldots \wedge x)$ under the b subtree and so on. However, doing so will not allow us to efficiently locate this materialized view, since again, there are an exponential number of sub-views that are potential parents of this clause.

We solve this tree-imbalance problem as follows: Let $\mathcal{E}$ represent the clause $(a \wedge b \wedge c \wedge \ldots \wedge x)$ which we want to add to the view tree. First, we deterministically pick a permutation $\mathcal{P}$ of $\mathcal{E}$ uniformly at random among all the possible permutations of $\mathcal{E}$. For example, the clause $(a \wedge b \wedge c)$ will be mapped with same $1/6$ probability to the $(a \wedge b \wedge c)$, $(a \wedge c \wedge b)$, etc. There are well known methods for generating random permutations, e.g. by exchanging array elements initialized with the array index, which can be used to generate random permutations deterministically. We then place the view $\mathcal{P}$ in the view tree such that the parent of $\mathcal{P}$ is the longest existing prefix of $\mathcal{P}$.

We illustrate this scheme in Figure 2. Suppose $a \wedge c \wedge e \wedge g$ deterministically maps to $g \wedge a \wedge e \wedge c$. This clause is now added to the view tree under its best current prefix in the tree, which is $g \wedge a \wedge e$. We discuss how the tree is maintained as new clauses are added below.

2.3 Maintaining the View Tree

The view tree must be updated to reflect new materialized views. This is done when a view is materialized using the procedure described in Section 2.2. The tree must also be updated when nodes holding already materialized views depart, or when views are simply discarded.

Tree integrity requires that the owner of a new node updates all attribute indexes corresponding to attributes of the new node. If the application requires this new node must be part of each resolved query immediately, then the node owner must also traverse each attribute subtree and update all existing materialized views before the node is added to the system. The view tree provides exactly the pointers that must be traversed for these updates. Further, the tree has the property that in order for a node to be updated, the entire path to the root must also be updated. Thus, if the update ever reaches a view that it does not need to update, it can discard the entire subtree under this view. For most applications, it is probably sufficient to only update the attribute indexes, and *not* update any of the materialized views. In this case, the materialized views should be periodically refreshed using its parent node and other appropriate view searches. For these applications, new nodes will not appear in a few queries immediately after they are added to the system.

We use a soft state refresh to maintain the integrity of the view tree. Child views periodically send heartbeat messages to their parent node in the view tree. If these messages are not received for a timeout period, the parent node simply discards state for the child view (and correspondingly its entire subtree). If a parent node is not alive, then the child re-inserts itself in the view tree.

Lastly, as views are added to the system, the child pointers need to be reassigned. Figure 3 shows a new clause, “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”.

3 Preliminary Results

This section presents preliminary results from simulations of the algorithms described above. We begin with a description of the data and query sources we used in the simulations along with the methodology we employed in deriving the input to drive our simulations.

3.1 Data Source and Methodology

We chose a random sets of documents from the TREC-Web data set as the source data for our experiments. We used HTML pages that exported the `keyword` meta-tag, and nominally used 64K different pages for each experiment.

We ran each experiment with 500,000 queries; this number was sufficient in all experiments for the caching behavior to stabilize. The queries were generated as follows: we first chose a representative sample of WWW queries from the publicly available `search.com` query database[1]. Unfortunately, the `search.com` query set does not provide an associated document set over which these queries would be valid: instead, 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.

Next, we use the distribution of keywords in the set of source documents to map keywords to each attribute. For multi-attribute queries, we generated the set of 10,000 most popular keyword digrams, trigrams, etc. and used these, uniformly at random, as the input query set for multi-attribute queries. Note that the popular 10,000 covered all possible multi-attribute queries with non-null results for our source data.

For each experiment, we use a "working set", which is a set of unique queries to which some fraction of overall queries are directed. Nominally, we used a working set of size 50,000 to which 90% of the queries were directed. 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.

We also model updates to the data. Specifically, we consider the cases when attributes are added from, deleted from, and changed in existing documents. New attributes, for both addition and updates, are chosen using the original keyword distribution generated from the complete source data set. For deletion, attributes are selected uniformly at random. It is not clear exactly what the rate for such updates should be.

Our primary metric is the number of tuples intersected when answering multi-attribute queries. 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.

Both single- and multi-attribute indexes have to be updated as data items are added, updated, or removed from the namespace. We present results for 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.

[1]We were also given access to 32K WWW queries by the IRCache project, and the query characteristics of the IRCache and `search.com` queries were comparable. We ran simulations with both data sets with similar results; we only present results from the (larger) `search.com` query set here.

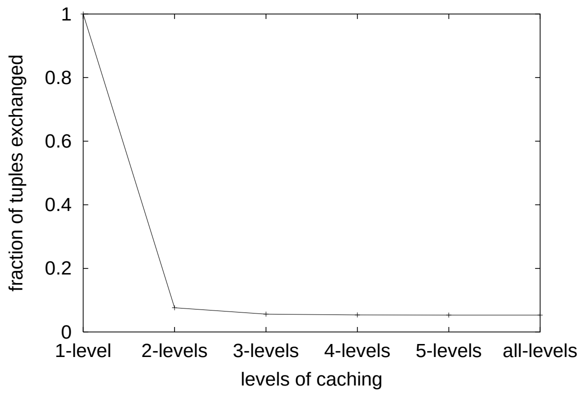

Figure 4: Caching benefit by level

3.2 View Tree Results

In our first result, we investigate the benefit of maintaining the view tree by simulating keyword searches over this data set. These experiments were run using our base parameter set: there were 500,000 total queries, 90% of which go to a working 50,000 queries, and the rest are chosen uniformly at random. In Figure 4, we show how the number of tuples intersected decreases as multi-attribute query results are cached. The x-axis shows the maximum depth of the view tree (e.g. depth 3 implies results of only two and two-attribute queries are cached). The y-axis is a measure of the benefit from maintaining the view-tree, and shows the normalized number of tuples transferred for each level of caching. For the normalization, we use the number of tuples transferred for the single attribute indexes only case as unity. From the plot, it is clear that there is potentially an immense benefit to maintaining a view tree: keeping only the second-level indexes reduces the number of tuples transferred by 92%. Extra levels of caching further reduces the intersection overhead by a further 30% compared to the two-level only caches (to less than 95% of the original). In terms of the actual number of tuples, the single-attribute indexes required 486M tuples to be intersected in total (972 tuples intersected/query). The two-level and all caches required 37M and 26M tuples intersected respectively. Recall that for exact queries, these numbers represent an upper bound on the number of tuples that have to be transferred over the network; for approximate methods, the number of tuples intersected is a lower bound on the amount of processing per query.

Obviously, the number of tuples that must be stored at each host increases as more indexes are maintained. For these experiments, the amount space required increased by a factor of

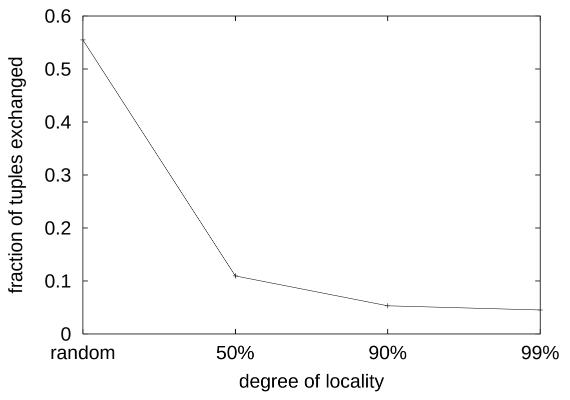

Figure 5: Effect of locality on caching efficiency

3.06 (0.6M tuples stored for the single attribute indexes vs. 1.84M tuples stored in 55K different multi-attribute caches for caching all levels) in the worst case. The two-attribute queries themselves require 1.5M tuples in 30K caches, which, in terms of tuple storage, is about 70% of all the multi-attribute queries. The two attribute indexes consume most space since the indexes with more attributes tend to be smaller as it is less likely that documents will export the same three or more keywords.

An interesting footnote from our experiments was that the number of cache hits monotonically *decreases* (by about 25% as the caching level is increased from 2 to all) as the level of caches is increased. This is because with more caches, queries have a higher probability of a direct cache hit in a cache with many attributes; however, with only a subset of caches, the popular subqueries get used over and over again.

3.3 Query Locality

It is important to understand exactly how sensitive our results are to the degree of locality in the query stream. In our base experiments, we choose a working set of 50,000 queries, and direct 90% of all queries this working set. In this experiment (Figure 5), we quantify the benefits from result caching as the degree of locality is varied. The x axis represents the degree of locality: specifically, it shows what fraction of the queries were chosen from the working set of 50,000 queries[2]. The rest of the queries are chosen uniformly at random. For the leftmost set of points, all queries are chosen uniformly at random, i.e. there is no locality in the query stream. The y-axis shows the normalized benefit from maintaining a full view tree (i.e. it is the ratio of the number of tuples intersected with and without a view tree).

We should note that the *number* of caches created for different localities vary by an order of magnitude (288K caches for no locality vs. only 29K caches when 99% of the queries are directed to the working set). There are only 64K cache hits in case of the random queries, thus, the view tree is of limited use when the query stream is random. However, even the random set of queries do get some benefit from the view tree (50% reduction in the number of intersected tuples). This is somewhat counterintuitive, since there is no locality in the queries. The benefit derives from the observation that even though the queries themselves are uncorrelated, there is correlation in the individual *attributes* that make up the queries. The attributes are chosen using their distribution in the exported keywords, and thus, the view tree becomes useful. We did conduct experiments in which both the queries and attributes are chosen uniformly at random. As expected, in this case, there is no benefit from the view tree; further, the vast majority of multi-attribute queries result in zero or a very small number of tuples.

[2]We have experimented with other working set sizes, and these results are representative.

3.4 View Maintenance: Updates

In the previous two results, we have shown that result caching reduces query resolution overhead, and is relatively robust as long as there is reasonable locality in the input. In this section, we show that the update overhead of the view tree is essentially negligible for almost all deployment scenarios. In Table 1 we report the number of caches updated, and the number of distinct nodes visited, over two 100,000 query simulations with different input distributions, and query vs. update ratios. “Update hops” includes the cost of finding caches via the view tree. In these simulations, attributes (keywords) are added or deleted from documents, and these updates are reflected in *every* cache that holds a pointer to this document. In the worst case scenario, there are only ten queries in the entire system before an attribute is changed in some document. Note that in the first row (1:10), there are 10,000 updates during the course of the simulation. The effect of locality in the query stream is apparent: query locality reduces the number of caches, but *increases* the overall update overhead to a popular document now has to update multiple caches. Specifically, the updates with a random query stream results in, on average, 3 different caches being updated, while the query stream with locality has to update approximately 10 different caches, again on average. Thus, in the worst case, each update would cause a single tuple to be transferred to ten different hosts. In practice, overheads will be lower since updates will be batched, and the update to query ratio is likely to be very low for most data.

4 Previous Work

Most closely related to this work are other efforts to provide a search infrastructure over peer-to-peer systems. Harren et al. [1] propose traditional relational database operators on top of DHT-based systems to resolve queries. Reynolds and Vahdat [2] discuss a search infrastructure using distributed inverted indexing. Each entry corresponds to a keyword and the set of documents that contain the keyword. Each node

Update:Query	90% locality		Random Queries	
ratio	#Caches	#Update Hops	#Caches	#Update Hops
1:10	28244	106683	62607	33758
1:100	28253	10774	62619	3430
1:1000	28255	923	62608	296

Table 1: Update overhead for 100,000 queries. 90% locality refers to 90% of the queries directed to the 50K working set.

in the system is responsible for all keywords that map the node. Tang et al. [4] argue for context-based and semantic-based text searches on top of DHTs. They extend vector space model (VSM) and latent semantic indexing (LSI), and support keyword-based queries. Annexstein et al. [5] argue for combining text data to speedup search queries, at the expense of more work done attaching/ detaching a node to a super-node. The indexes are kept as suffix trees. Kubiatowicz [3] proposes to use transactional query support. The set cover problem and a greedy approximation algorithm is discussed by Cormen et al. [10].

5 Summary and Discussion

This paper presents the design of a scalable and efficient search infrastructure using a new structure called the view tree. Our preliminary results show that 1) use of result caches can eliminate the vast majority of tuples retrieved across the network for queries with multiple terms, 2) result caches are effective even with no locality in the query stream (but with locality in the distribution of attributes across documents), and 3) update cost should be relatively insignificant.

In addition to testing with a wider variety of input data, we need to address a number of significant issues before a real system can be designed. These issues include result cache placement policies, result cache replacement policies, fault tolerance, availability, and interactions with the underlying DHT system. Finally, both application consistency requirements and available system resources should ideally be taken into account by view maintenance policies. Cached results, and even the basic distributed index scheme, also open up a wealth of new security problems.

References

[1] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica, "Complex queries in dht-based peer-to-peer networks," in *The 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[2] P. Reynolds and A. Vahdat, "Effi cient Peer-to-Peer Keyword Searching," *unpublished*.

[3] 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 Languages and Operating Systems*, Cambridge, MA, November 2000, pp. 190–201.

[4] C. Tang, Z. Xu, and M. Mahalingam, "pSearch: Information retrieval in structured overlays," *SIGCOMM Computer Communication Review*, vol. 33, no. 1, January 2003.

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

[6] B.H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, 13(7):422-426, 1970.

[7] M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed INGRES," *IEEE Transactions on Software Engineering*, vol. 5, no. 3, pp. 188–194, May 1979.

[8] P. Slavik, "A tight analysis of the greedy algorithm for set cover," in *ACM Symposium on Theory of Computing*, Philadelphia, PA, May 1996, pp. 435–441.

[9] Y. Minsky, A. Trachtenberg, and R. Zippel, "Set reconciliation with nearly optimal communications complexity," in *IEEE International Symposium on Information Theory*, Washington DC, June 2001.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 1997.

[11] I. Wegener, *The Complexity of Boolean Functions*, John Wiley & Sons Ltd., and B.G. Teubner, Stuttgart, July 1987, ISBN: 0-471-91555-6.