Distributed Cache Table: Efficient Query-Driven Processing of Multi-Term Queries in P2P Networks*

Gleb Skobeltsyn
EPFL, Lausanne, Switzerland
gleb.skobeltsyn@epfl.ch

Karl Aberer
EPFL, Lausanne, Switzerland
karl.aberer@epfl.ch

ABSTRACT

The state-of-the-art techniques for processing multi-term queries in P2P environments are query flooding and inverted list intersection. However, it has been shown that due to scalability reasons both methods fail to support full-text search in large scale document collections distributed among the nodes in a P2P network. Although a number of optimizations have been suggested recently based on the aforementioned techniques, little evidence is given on their scalability. In this paper we suggest a novel query-driven indexing strategy which generates and maintains only those index entries that are actually used for query processing. In our approach called *Distributed Cache Table*[1] (DCT) we suggest to abandon the difference between data indexing and query caching, and to store result sets (caches) for the most profitable queries. DCT employs a distributed index to efficiently locate caches that can answer a given multi-term query and broadcasts the query to all the peers only if no such caches were found. Evaluations on real data and query loads show that DCT converges to a high cache-hit ratio and indeed offers a large-scale distributed solution for storing and *efficient* querying of vast amounts of documents in the P2P setting. DCT achieves two orders of magnitude improvement in traffic consumption compared to a standard distributed single-term indexing approach.

1. INTRODUCTION

P2P systems have been successfully employed by global-scale file-sharing applications, proving that such systems are capable of storing vast amounts of data. However, to make P2P systems a viable architectural alternative for information retrieval and database-oriented applications, support for expressive queries is required. An arbitrary complex query can be answered by broadcasting it to all peers in the network, since each peer receiving the query, can locally evaluate it and return its contribution to the overall result set. However, this comes at the expense of high bandwidth consumption. An alternative solution are structured P2P systems, as they typically offer logarithmic search complexity in the number of participating nodes. Although the use of a distributed index (typically a DHT) enables more efficient query-processing, it requires sophisticated index design because single-term indexing solutions, even when carefully optimized, generate unscalable search traffic [24].

In this paper we concentrate on a problem of efficient processing of multi-term queries over a corpus of text documents placed in a large-scale P2P distributed storage. As the number of peers in the network can be large, the use of an unstructured network is unfeasible due to high bandwidth consumption induced by frequent broadcasts. Thus, we employ a standard DHT approach to associate queries to their result sets in a non-trivial fashion. Our approach is motivated by the observation that a large number of index entries may never be queried and therefore the maintenance of such entries is unnecessary. We employ a novel indexing/caching strategy, Distributed Cache Table (DCT), for efficient processing of multi-term queries that is driven by the *query load*.

DCT populates the storage space provided by participating peers with result sets (caches) for carefully chosen queries and uses this data to answer further queries. Each cache stores a list of document digests[2] and hence can be used to resolve any query if its result set is contained in the list. Peers maintain those caches which are frequently used to answer queries and consume little space[3]. DCT performs an adaptive selection of queries to cache, based on the monitored query statistics taking into account limited storage capacity with the goal of minimizing the number of cache-misses. In particular, each peer runs a greedy algorithm leading to a global quasi-optimal cache selection. Therefore, DCT adopts a *query-adaptive indexing strategy*.

To get a general idea about our approach consider a scenario where N peers form a DHT-based P2P network. Each peer shares some of its documents with other peers and is

*The work presented in this paper was (partly) carried out in the framework of the EPFL Center for Global Computing and supported by the Swiss National Funding Agency OFES as part of the European projects BRICKS (507457) and ALVIS (002068).

[1]By analogy with Distributed Hash Table (DHT)

[2]A document digest contains an unique document identifier and a list of terms extracted from the document.

[3]The reasoning behind the cache size restriction is related to limited storage and traffic consumption and refers to the approach that indexes discriminative term sets (they are queries in our case) associated with result sets of constrained size [13].

able to search documents located at other peers by issuing multi-term queries. A straightforward solution is to broadcast queries to all peers, so each peer can evaluate each query locally.

Let us assume now, that every peer π can provide a limited storage space s_π. The whole network then has $S = \sum_{i=1}^{N} s_{\pi_i}$ storage capacity that peers utilize to store query caches. A cache for the query q stores its *result set* RS_q, which contains documents digests for all documents satisfying q. They are sufficient for query filtering, i.e., we can locally answer q using $RS_{q'}$ if $RS_{q'}$ contains RS_q. Now, to answer a query q we first try to find (at least one) cache which can answer q, and issue a broadcast only if no such cache was found. To entirely benefit from caching, DCT exploits *query subsumption*. Hence, we are interested in locating any cache which *contains* the result set of q. Thus, the DCT network evolves in time to a large-scale distributed cache driven by the query load, avoiding maintenance of (almost) never used data and employing broadcasts only for rare queries.

DCT employs a distributed ranking mechanism described in [13]. When a stored result set (cache) is used to answer a query, it processes the query locally and returns only k top ranked documents. The next portion of documents can be supplied on demand. This mechanism significantly reduces the traffic consumption.

In summary, the main contributions presented in this paper are the following:

- We introduce a novel query-driven indexing strategy for multi-term queries in a P2P environment that is based on the query subsumption property;
- We perform experiments with real query traces that show a high number of subsumption dependencies in realistic query loads that are beneficial for our approach;
- We achieve a significant overall traffic reduction compared to the distributed single-term indexing approach.

The paper is organized as follows: We position our approach with respect to related work in Section 2. We describe the caching strategy in Section 3, in particular, a distributed meta-index in Section 3.1, followed by the cache management discussion in Section 3.2. We discuss load balancing issues in Section 4. Simulation results are presented in Section 5, followed by the conclusion in Section 6.

2. RELATED WORK

Full-text P2P search has been investigated for both unstructured and structured P2P networks. Search techniques in unstructured networks are usually based on broadcasts, thus suffering from high bandwidth consumption. Hence approaches based on random walks [23] and hierarchical network solutions [11] have been proposed to reduce the generated traffic in a P2P network.

Structured networks are more difficult to maintain, but offer considerably better search efficiency compared to unstructured networks by utilizing a distributed index (typically a DHT). DHTs provide efficient single-term lookups by hashing terms into keys. Assume the distributed index stores posting (inverted) lists for all terms found in a document collection, then a multi-term query is usually resolved by intersecting these lists for all terms in the query. However, this approach faces significant scalability problems caused by high traffic costs required for intersecting large posting lists. For example, [15] and [18] have proposed top-k posting list joins, Bloom filters, and caching as promising techniques to reduce search costs for multi-term queries. However, a recent study [24] shows that single-term indexing is practically unscalable for web sizes even when sophisticated protocols using Bloom filters are combined to reduce retrieval costs.

The costly intersection can be avoided if inverted lists store so-called document digests, sufficient for local query answering. In this case, a query can be resolved from a posting list for any of its terms. We refer to this approach as single attribute dominated, as defined in [3]. Although it insures that the traffic caused by query processing is low, the size of the index becomes extremely large, reaching storage capacity limits. Moreover, populating the index causes very high traffic consumption. Therefore, this approach is typically used by smaller scale applications such as [3, 1, 20]. The paper [19] uses the single attribute dominated approach in P2P information retrieval. The authors designed a hybrid indexing scheme which sacrifices the search quality in order to reduce the index size.

However, the techniques presented above assume that all data available in the network has to be indexed. This assumption results in generation and maintenance of a large number of index entries which may (almost) never be utilized. In contrast, our adaptive indexing solution generates the index "on-the-fly", driven by the query load. It adapts to the global storage capacity provided by the peers[4], therefore limiting the indexing traffic; at the same time it achieves low traffic consumption during query processing.

Probably the closest approach to ours with respect to the caching strategy is [2]. The authors employ a similar idea, though the indexing part is done differently, no load balancing is considered and no explicit results showing query processing costs were reported. The paper [10] suggests to avoid maintaining large posting lists by complementing index-based query processing with broadcasting. In contrast to our approach, the authors suggest using flooding mechanisms to answer popular queries, and leverage indexing only for rare queries.

Caching of query result sets as the performance improvement for an existing distributed index is used by many P2P indexing approaches. For example, [17] and [7] utilize caching for efficient processing of XPath queries. [9] and [16] employ caching to improve the search efficiency while processing range queries.

DCT is in a way similar to the Freenet approach [4] since it adopts document caching along the search path of a query, however the latter pursues the different goal of network clustering.

Finally, [12] suggests a similar idea to our query-driven indexing, but applied to a local XML database for answering XPath queries. Their results conform to our observations about the high potential of cache-based query processing.

3. INDEXING AND CACHING STRATEGY

Let us assume a network of N peers, π_i, $i \in 1..N$, where each peer hosts a part of the document collection D_{π_i} and

[4]In fact, selection of a storage capacity provided by a peer regulates the bandwidth consumption required for indexing.

issues queries from a local query load L_{π_i}. Therefore, $D = \bigcup_{i=1}^{N} D_{\pi_i}$ is the global document collection and $L = \bigcup_{i=1}^{N} L_{\pi_i}$ is the global query load.

Let us define a superset $T = \{t_1, t_2, .., t_m\}$ as the vocabulary consisting of all single terms found in the global query load L. Then, a query $q \in L$ is defined as $q = \{t_1, t_2, .., t_n\}$ and is a subset of T, $q \in 2^T$. The number of terms in q is denoted as $|q| = n$. Similarly, we define a document $d \in D$ as $d = \{t_1, t_2, .., t_r\}$ which is also a subset of T: $d \in 2^T$. Essentially, we simplify the representation of an original document d_0 by intersecting the set of all terms contained in d_0 with T, $d_0 = d \cap T$, and therefore ignore the terms contained in the original document which do not appear in the query load.

A query q matches a document d iff $q \subseteq d$. The result set RS_q for the query q is the set of all documents matching q, $RS_q = \{\forall d_i \in D \mid q \subseteq d_i\}$.

Now we define the *query subsumption* relation as follows: A query q' subsumes a query q when all terms in q' are also contained in q, i.e., $q' \subseteq q$. Obviously, $q' \subseteq q$ implies $RS_{q'} \supseteq RS_q$. In other words a query q can be answered by postprocessing of the result set associated with q'.

The set of all possible queries over T can be represented as a *lattice* of the size $2^{|T|} - 1$. Each lattice node corresponds to a query, and the whole lattice models the set of all potential queries over T that might appear in the query load L. For example, a lattice generated for the vocabulary of four terms $T_{abcd} = \{a, b, c, d\}$ is shown in Figure 1.a. An arrow from a query q_1 to a query q_2 reflects the subsumption relation $q_1 \subseteq q_2$. Figure 1.b highlights all descendants of nodes a and cd, referring to all the queries that are subsumed by the queries a and cd. Indeed, all queries containing either the term a or both terms c and d can be answered from the two result sets RS_a and RS_{cd}.

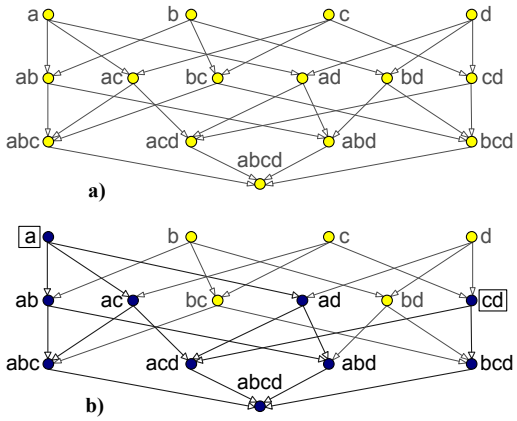

Figure 1: Query subsumption: a) a,b,c,d lattice; b) lattice with the queries "a" and "cd" being cached.

Each peer has a certain storage capacity and uses it to store carefully chosen result sets. In fact, a peer π caches result sets for certain queries from its local query load L_π and advertises it to other peers using a distributed *meta-index*. Therefore, to answer a new query, another peer may lookup the location of existing caches that may resolve the query as it is described in Section 3.1. Furthermore, as the peer storage capacity is limited, each peer runs a greedy cache-selection algorithm as described in Section 3.2.

3.1 Meta-index

The meta-index facilitates efficient lookup of the result set (cache) locations for a given query. Formally, given q we need to obtain a result set RS_q by locating at least one cache $RS_{q'}$ such that $RS_{q'}$ *contains* RS_q ($RS_{q'} \supseteq RS_q$). In other words, we are interested in locating a cache for q', such that $q' \subseteq q$. To locate relevant result sets, we introduce a distributed meta-index which stores links to actual cache locations. Given a query q, the meta-index returns a list of tuples $\{q_i,\ uri(RS_{q_i})\}$ for the queries which are cached and subsume q. A *random* tuple from the received list is selected and the query q is forwarded to the peer storing the chosen cache. This peer processes q locally and returns the list of documents matching q. If no caches were located by using the meta-index, i.e., the query can not be answered from the cache, it is broadcasted to all the peers that evaluate the query against their local document collections and send the answers to the originating peer. Since peers participate in a DHT, we can use the "shower broadcast" technique [6], which insures that each peer is visited only once. To answer a query q, $O(N)$ messages have to be send to notify all peers that generate $|RS_q|$ records of traffic while answering, where $|RS_q|$ denotes the number of records in the result set of q.

The meta-index is implemented using the standard *put/get* functionality offered by the DHT. Given a cache $RS_{q'}$ physically stored at $uri(RS_{q'})$ an *advertise* operation is performed by inserting a tuple $\{q',\ uri(RS_{q'})\}$ at the peer responsible for the $key = h(t_r)$, where $h()$ denotes the DHT's hash function and $t_r \in q'$ is a *randomly* chosen term from q'. Therefore, the advertise operation requires one *put* message to be send with $O(\log N)$ overlay hops.

Given a query q to be answered, the meta-index lookup operation is performed in the following way: $n = |q|$ messages containing the original query q are sent to the n peers responsible for $h(t_1),\ h(t_2),\ ..,\ h(t_n)$, where $t_i \in q, \forall i \in 1..n$. Each peer responds with a list of cached result set locations for queries that subsume q. Therefore, the cache lookup operation requires n *get* messages to be send with $O(n\ \log N)$ overlay hops.

Section 3.2 introduces the cache management and explains how a peer can locally decide which caches have to be created or dropped leading to the quasi-optimal utilization of the overall network storage capacity S, thus reducing the number of broadcasts for the current query load.

3.2 Cache management

Having defined the meta-index, we can formulate the problem of finding an optimal set of caches in the network that maximize the number of cache-hits for a given query load and a P2P network with a constrained global storage capacity distributed among the participating peers.

Each query q in the global query load L is assigned a probability p_q of being queried. We assume that the result set sizes of all queries from L are known: $|RS_q|,\ \forall q \in L$ denotes the number of documents in RS_q.

We denote the set of cached queries as $\Omega \subseteq L$. To store (cache) result sets for all the queries in Ω, the following global storage capacity is needed (measured in the number of documents): $S_\Omega = \sum_{\forall q_i \in \Omega} |RS_{q_i}|$.

Our goal is to utilize the available storage as efficiently as possible which means to minimize the number of broadcasts or maximize the number of *cache-hits*. We denote a function $cachehit(q)$ as follows:

$$cachehit(q) = \begin{cases} 1, & \exists q' \in \Omega,\ s.t.\ q' \subseteq q; \\ 0, & otherwise. \end{cases}$$

Therefore, the cache optimization problem is to find a set Ω containing queries to be cached that maximizes the number of cache-hits:

$$\Omega = argmax \sum_{\forall q_i \in L} cachehit(q_i)\ p_{q_i},$$

having a storage constraint:

$$S_\Omega = \sum_{\forall q_i \in \Omega} |RS_{q_i}| \leq S_0.$$

The stated optimization problem is similar to the well-known 0/1 knapsack problem [8] (which is known to be NP-complete), applied to all queries from L. The increased complexity of the cache optimization problem compared to the knapsack problem is caused by the fact that we cannot assign constant profits to queries (items) due to the subsumption-related inter-dependencies between the queries. Furthermore, as the query load is dynamic, we are rather interested in a decentralized algorithm which leads to a quasi-optimal solution.

Indeed, each peer has to decide locally on a set of queries it caches to fill in its available storage. A peer pursues a greedy cache-selection strategy by deciding to cache queries such that their estimated *profits* are high. We define a max profit of the query q as: $profit_{max}(q) = \frac{g_q}{|RS_q|}$, where $g_q = \sum_{\forall q_i \subseteq q} p_{q_i}$ refers to the probability of the query q being utilized to answer any query from L if no other caches are available. However, since there could be more then one cache capable of answering a given query due to the subsumption, the actual profit is lower and depends on the existing caches in the network.

An estimate of the query profit can be obtained from the statistics as $profit(q) = \sum_{\forall q_i \subseteq q} \frac{bfreq(q_i)}{|RS_q|}$, where $bfreq(q_i)$ is the number of broadcasts of q_i (because no caches subsuming q_i were found) observed *recently*. In this formula we can distinguish an absolute frequency $af_q = bfreq(q)$ of the query q being queried and a subsumption frequency $sf_q = \sum_{\forall q_i \subseteq q} bfreq(q_i)$. The latter one counts all queries subsumed by q including q itself for the current state of the network. Obviously, $af_q \leq sf_q$. After we defined the subsumption frequency, the query profit can be finally expressed as:

$$profit(q) = \frac{sf_q}{|RS_q|}.$$

DCT peers perform local and isolated maintenance of the global query statistics: each peer has a global view (by listening to broadcasts) on the locally selected subset of queries it monitors. We restrict this monitored subset to the set of popular queries this peer used to submit in the past. The advantage of this mechanism is that approximate result set sizes are already known from the history. The statistics module counts recent absolute and subsumption frequencies for each query. When a peer caches a new query, it advertises the new cache in the meta-index as described above.

For every existing cache similar statistics are maintained measuring its absolute and subsumption cache-hit values, in order to drop it if more profitable caches were found.

Following a greedy strategy a peer can create a new cache if there is enough space available or the required amount of space can be released by dropping caches with lower profits. Hence, each peer locally selects the most profitable caches for the available local capacity. Therefore, if the query load is static, the greedy strategy ensures that the resulting cache-hit can only increase or remain the same.

Due to the multiple subsumption dependencies between queries, caching or dropping a query might substantially influence statistics maintained for other related queries. We argue, that the presented strategy, though simple, gracefully adapts to the P2P network instability and changes in the query load. Indeed, the cost of adding a cache is only $O(\log N)$ overlay hops (needed to modify the meta-index), while dropping a cache causes only one extra message to be sent. In case of a peer failure all caches it stored or indexed become unavailable, causing broadcasting the associated queries. Thus, other peers will probably cache them if the profits are high enough.

However, a popular query q_0 might not be cached if it is associated with a large result set because its profit could be relatively low. In this case, DCT will react by caching popular derivatives of q_0 (queries subsumed by q_0) if needed. However, the meta-index would report a cache-miss for q_0 itself, and it would have to be broadcasted every time. To solve this problem we suggest caching only top-k result of q_0. Obviously, this "top-k" cache can not be utilized to answer any other query except for q_0. Its profit can be estimated as

$$profit_{topK}(q_0) = \frac{af_{q_0}}{k}.$$

Recall, that af_{q_0} denotes the absolute query frequency of q_0 being queried. The constant k reflects the maximum number of records the majority of the users would browse[5].

A DCT's top-k selection algorithm chooses the best type of cache for each query, by comparing the estimations of profits calculated for the top-k and full case. If adding a new full cache fails, the second attempt is made as top-k. A top-k cache can be switched back to a full cache by issuing a broadcast if the profit of the full cache is higher. Alternatively, when a full cache is about to be deleted it can be switched to a top-k instead.

The presented strategy facilitates the distributed selection of caches being constrained with the available storage capacity in the network and leads to a quasi-optimal solution with respect to the minimization of the traffic consumption. Furthermore, utilizing top-k caches significantly reduces the number of cache-misses for the queries associated with large result sets and further decreases the traffic consumption.

3.3 Example

Let us illustrate the approach by an example. Initially, all queries are broadcasted and each peer has to evaluate each query over its local document collection. A peer π joins the network and starts processing broadcasts and issuing its own queries. π maintains statistics about the queries from its local query history, for example:

$Query$	$\lvert RS_q \rvert$	af_q	sf_q
cd	500	5	50
a	5000	98	100
ab	2000	21	23

[5]Note, that our top-k caching strategy follows the original idea of indexing discriminative term sets [13]

The statistics table stores information about the most frequent queries from π's local query history. Recall that af_q and sf_q denote the absolute and subsumption frequencies respectively. $|RS_q|$ is estimated locally, since the result set for q was obtained by π before.

Assume the query cd is issued at π. The peer computes the full and top-k profit estimates as $profit(\text{“}cd\text{”}) = \frac{sf_{cd}}{|RS_{cd}|} = 50/500 = 0.1$ and $profit_{topK}(\text{“}cd\text{”}) = \frac{af_{cd}}{k} = 5/250 = 0.02$, if $k = 250$ is chosen. Then it checks if there is enough storage space to cache cd as a full cache. If not, π compares $profit(\text{“}cd\text{”})$ with profits of already existing caches and drops some of them if needed to store the result set of cd. Alternatively, it can consider caching the query as top-k, which would be the case for query a for example.

Assume π is caching cd. It issues a broadcast and stores the obtained result set. To make the cache available for other peers in the network, π generates a key: $key = h(\text{“}c\text{”})$ (or $key = h(\text{“}d\text{”})$) and inserts a tuple $\{\text{“}cd\text{”}, address_\pi\}$ into the meta-index using the key. The tuple is routed to the peer π_{key} responsible for the key and π_{key} stores the tuple. Assume also, that another peer caches a (top-k) result set for the query “a”.

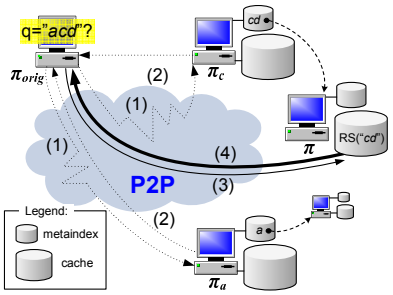

Figure 2: Query processing example

Figure 2 shows how the 2-step query processing is performed. A peer π_{orig} submits the query $q =$ “acd”. First, the meta-index is searched for available caches. To do so, 3 messages containing q are routed to the peers responsible for $h(\text{“}a\text{”})$, $h(\text{“}c\text{”})$ and $h(\text{“}d\text{”})$ respectively (1). π_a, π_c and π_d browse their meta-index tables and send back the lists of relevant caches (2). Please note that since the query a is cached as top-k, it cannot be used to answer acd. Hence, the information that the query cd is cached at $address_\pi$ received from the peer π_c will be used. The originating peer requests π to answer the original query (3). π responds back with the answer (4). In case no caches were found in the meta-index, a broadcast would be used to answer the query.

4. LOAD BALANCING

Since our approach is based on caching popular queries, peers can suffer from certain load imbalances due to both non-uniform meta-index lookup requests and uneven cache utilization. In this chapter we argue that load imbalance caused by these factors can be efficiently tackled without substantial performance degradation. In the following we discuss load balancing issues for both cases in details.

4.1 Meta-index load balancing

We argue that due to the small size of the data stored in the meta-index and a certain randomization in the advertise operation, almost no explicit load balancing of the meta-index is necessary. However, peers that are responsible for the most popular terms can receive a large number of incoming requests, which can be avoided by handling such terms in a special way, as explained in [5]. First, these terms are marked as popular and the index information involving them is moved to alternative locations if possible, since a query $q = t_1..t_n$ can be indexed on any of the n peers. Then, the terms are advertised to the forwarding peers[6] as popular so these peers can take part of the load caused by the popular term. Thus, subsequent requests will not reach the original peer, but will be pruned on the way, leading to a better load distribution.

Indeed, our evaluations show, that only several top popular terms cause very high meta-index lookup load. Hence, the solution proposed above would split the load among neighboring peers, avoiding meta-index lookup hot-spots.

4.2 Cache access load balancing

Balancing of load caused by resolving queries from caches is more crucial due to the high traffic it creates to supply query results compared to the meta-index lookup. However, our evaluations show that only several top popular caches are accessed very often and cause serious load imbalance.

Thus, standard DHT replication mechanisms can be successfully utilized to relieve overloaded peers. Furthermore, our preliminary investigation shows that more sophisticated load balancing can be embedded into the DCT algorithms. We leave the detailed analysis of the cache load balancing mechanisms for future work.

5. EXPERIMENTAL RESULTS

In this section we report experimental results obtained by using our DCT simulator implemented in Java. The data collection used in the experiments is the Wikipedia [21] document collection (6Gb XML dump of the core English Wikipedia from May 2006 available at [22]) containing 3M pages.

We used two real Wikipedia query load traces from August and September 2004. Both of them have very similar properties, hence we summarize only those of the August trace. From the total of 4.6M queries, there are 1.3M unique queries. There are 0.5M queries occurring at least twice and 250K at least three times in the query load. The queries contain 160K unique terms while the average number of terms in a query is 2.6.

Both the Wikipedia document collection and the query loads were preprocessed by applying the Porter stemmer [14]. Before performing the experiments we obtained query result set sizes for all the queries: First we built an in-memory single-term index for all terms appearing in the query loads and then we computed cache sizes for each query by intersecting posting lists for its terms.

5.1 Simulation setup

The DCT simulator creates a number of peers with a predefined available cache capacity. It iteratively chooses random peers to generate queries and simulates the distributed query processing using the algorithms defined in Section 3.

[6]Forwarding peers are peers that have π_p in their routing tables, where π_p is the peer responsible for a popular term

A query generator selects a real query from a query trace following the trace distribution.

We decided to limit the available storage capacity per peer to 200K records and fixed the top-k cache size to 250 records. As mentioned in Section 3.2, we monitor only recent query statistics, hence we selected a reasonable size of 200K broadcasts for the history window. In other words, the query statistics are maintained for the period which covers the last 200K broadcasts. Finally, a query can be cached only if it was already answered before, since its result set size has to be known.

In our experiments we measure three values: *CacheHit*, *SubsumHit* and *TopKHit*. *CacheHit* reflects the fraction of queries that were answered from caches, therefore the remaining $(100 - CacheHit)\%$ queries were answered using a broadcast. A query q can be answered from a top-k cache producing a *TopKHit*. Alternatively, we have a *SubsumHit* if q was answered using $RS_{q'}$ cache and the issued query q is different from the cached query q' (formally, $q' \subset q$). Obviously, $CacheHit \geq TopKHit + SubsumHit$.

5.2 Storage capacity

In this experiment we explore how much capacity is needed to ensure a reasonable cache-hit ratio. We vary the number of peers N, thus changing the overall network capacity as $N \times 200K$ records. Figure 3 plots the maximum *CacheHit*, *SubsumHit* and *TopKHit* achieved after the network converges to a stable state by processing 4.6M queries.

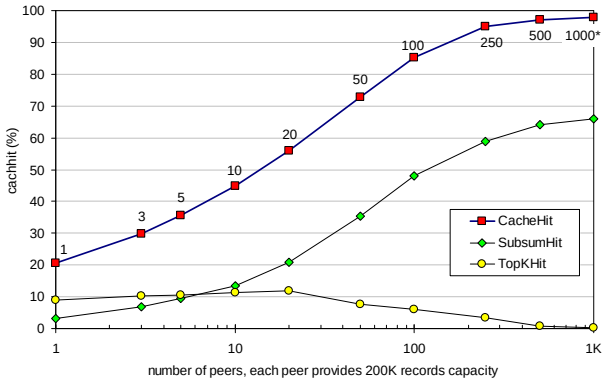

Figure 3: Max achieved *CacheHit*, *SubsumHit* and *TopKHit* with different number of peers.

Figure 3 shows that a high cache-hit value can be achieved with relatively low storage capacity, e.g., DCT with 100 peers storing 200K records each, converges to 85% cache-hit. Another observation which we make from this plot is that top-k caches are extensively used when available storage space is limited (when the number of peers is below 20), whereas more and more queries are answered via subsumption as the capacity grows. Having high subsumption rate leads to a more robust network as we will see in a stress test experiment in Section 5.4.

When the DCT network contains 1000 peers (marked with the asterisk) only 71% of the available capacity is utilized, thus the achieved cache-hit of 98% is the maximum for our experimental setting. The remaining 2% are those queries that were not subsumed by any cached query and were asked only once, so caching had no impact on them. We achieved such low cache-miss ratio due to the high subsumption rate. Indeed, our simulations show that if peers have infinite capacity, but store only top-k caches, the maximum cache-hit that can be achieved is only 82%. This number can easily be obtained from the query load statistics. Recall, that out of 1.3M unique queries only 500K were repeated at least twice. Hence caching has no impact on 800K "single" queries. Having 4.6M queries in total these 800K queries are exactly the remaining 18%.

5.3 Traffic consumption

The main goal of our simulation is to show that the proposed query-driven approach is a promising alternative to standard single-term indexing techniques in a P2P-IR scenario. In the following we will show that DCT reduces traffic consumption by two orders of magnitude when compared to the naïve approach, which indexes single terms in a structured P2P network.

We implemented the naïve approach based on a single-term index. For each query we first eliminate stop words (we used a list of 260 common English words). Then, we locate peers responsible for remaining terms in the same way it is done by DCT. The query is answered by conveying a posting list between the peers responsible for the query terms. Posting lists are practically intersected along the way until reaching the final peer that produces the answer to the query and sends only top-10 records to the query originator (more records can be send on demand).

We implemented two variations of the naïve approach: *naïve-random* and *naïve-sort*. The first one chooses terms and responsible peers in a random order, whereas the second one sorts the terms according to the sizes of their posting lists and contacts the peers in that order. We measure the traffic required to process a query as a number of records which have to be transmitted in the network. Please note, that assuming the term index is already available because it was produced beforehand during the indexing phase, the traffic required to process a query depends only on the posting list sizes of its terms. We computed the average traffic for the Wikipedia query loads and the data dump and obtained the following values after processing 4.6M queries:

Approach	August 2004	September 2004
naïve-random	37 370 rec./query	38 919 rec./query
naïve-sort	8 232 rec./query	8 903 rec./query
broadcast	7 920 rec./query	7 197 rec./query

The size of the generated single-term index was 240M records. However, the term index was build only for terms from the query load, whereas in practice this information is not available in advance. While standard IR approaches try to extract all probable terms from the documents during indexing, DCT relies on monitoring the query load for "on-the-fly" index construction.

As we can see the naïve approach performs poorly in terms of traffic consumption due to large posting list sizes which have to be transmitted. Surprisingly, the traffic consumption of the simple broadcast is even slightly better than the naïve-sort, although it requires propagating each query to all peers in the network (note, that our traffic computation ignores the control and routing messages sent between peers).

Figure 4.a shows how the *CacheHit*, *SubsumHit* and *TopKHit* values increase with the number of processed queries.

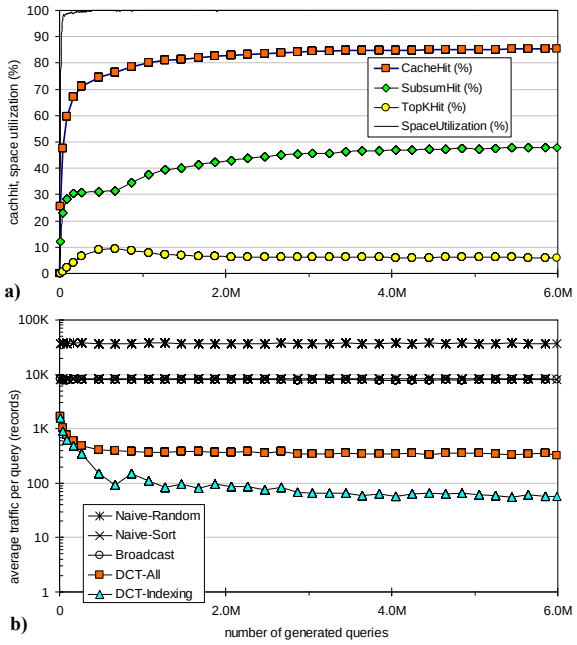

Figure 4: Network with 100 peers, 200K records per peer. a) Cache-hit; b) Traffic consumption.

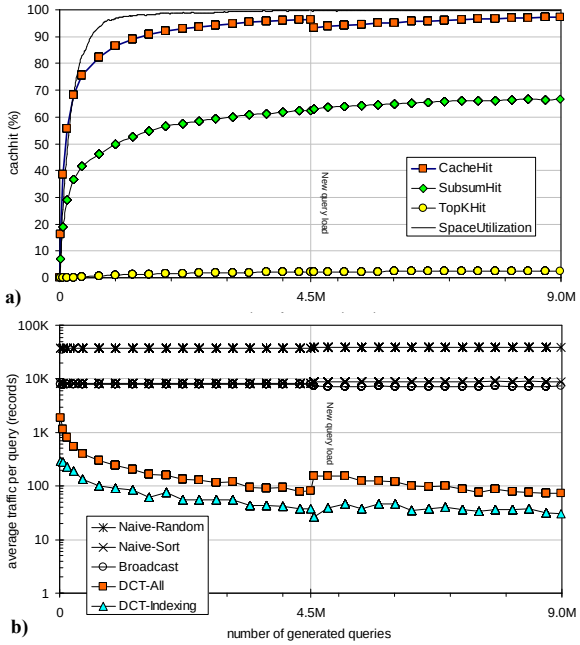

Figure 5: Stress test with 500 peers, 200K records per peer. a) Cache-hit; b) Traffic consumption.

The figure also plots the cache utilization curve, which shows that all available space is almost fully utilized after processing 30K queries. Hence, the following cache-hit increase is achieved by proper selection of caches. Finally, it can be observed that after enough statistics are gathered (after 200K broadcasts) the *TopKHit* starts decreasing while *SubsumHit* increases. It happens because some caches switch from top-k to full cache, based on the profit comparison.

Figure 4.b plots the average traffic per query generated by our approach (*DCT-All* curve). The *DCT-Indexing* curve shows the fraction of the traffic which was used to create new caches. Traffic consumption is reducing rapidly as DCT converges. We also output the naïve and broadcast traffic consumption for comparison. *DCT-All* curve is always below naïve and broadcast, moreover, the traffic consumed by our approach after DCT converged (approx. 140 records/query) is two orders of magnitude lower than naïve-sort or broadcast. As we will see in Section 5.4 the DCT traffic consumption decreases even further when the cache-hit increases. The ideal value of 10 records/query would be achieved if all queries are answered from the cache.

Despite of low traffic requirements, DCT causes more overlay messages to be sent. The difference comes from broadcasting the remaining $(100 - CacheHit)\%$ queries that cannot be answered from the cache. This price is paid however, for the much smaller index size due to its adaptivity to the query load. In terms of latency DCT requires $O(\log N)$ time to answer a query from cache plus additional $O(\log N)$ in case a broadcast is required. The naïve approach requires additional time to transmit (possibly) large posting lists which substantially increases the latency.

5.4 Stress test

We performed a stress-test: In the first part of the test we were generating queries from the August query trace and after 4.5M queries we switched to the September trace. Figure 5.a shows that DCT converges to the high cache hit ratio of approx. 98%, slightly drops when the query load changes and converges again. The change of the query load is quite smooth because of the high subsumption utilization. Figure 5.b shows the traffic consumption during the stress test. It can be observed that the traffic consumption drops to relatively low values and slightly increases when the query load changes. Finally, with 98% cache-hit ratio the traffic consumption reduces to approx. 75 records/query.

5.5 Load balancing

Figure 6 shows the peers' load obtained for a network consisting of 100 nodes and caused by answering queries from caches (Figure 6.a) and meta-index lookups Figure 6.b.

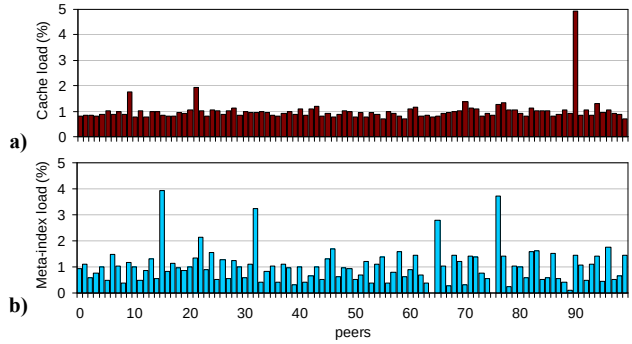

Figure 6: Load caused by a) cache access and b) meta-index lookups in the network of 100 peers.

It might seem that popular caches cause huge load imbalance. However, if a user requests only top-10 items and

taking into account unrestricted cache placement, which depends only on the peers' querying activities, the load is distributed almost evenly as it can be observed in Figure 6.a. The only problem is caused by several top-popular queries that create very heavy load. A native DHT replication mechanism can be used to solve this imbalance.

The meta-index service exhibits a certain load imbalance as shown in Figure 6.b, however it serves only index lookups that do not require large traffic transfers. Moreover, TCP/IP connections to the neighbors are maintained alive by the DHT, hence, the imbalance caused by the meta-index has low impact compared to the cache imbalance. Additionally, the mechanisms we proposed in Section 4.1 would help avoiding hot spots with low overhead.

6. CONCLUSIONS

In this paper we presented a novel query-driven indexing strategy for multi-term query processing with structured P2P networks. Our approach, called Distributed Cache Table (DCT), avoids maintaining rarely used index entries by adapting to the query load. DCT peers run the greedy algorithm leading to a quasi-optimal cache selection that maximizes the the global cache-hit ratio. DCT relies on the subsumption relation between queries while selecting a cached result set to resolve a query.

We performed an extensive experimental evaluation on real data and query traces that confirms the feasibility of our approach. The results have shown two orders of magnitude reduction in traffic consumption compared to the naïve single-term indexing approach.

We claim that DCT can be applied to the broader class of *conjunctive queries*. Such a query is expressed by a conjunction of atomic predicates: $q = a_1 \& a_2 \& .. \& a_n$, where the atomic predicate structure is application-specific, e.g., for the studied class of multi-term queries each atomic predicate is a natural language term. We leave exploration of additional benefits that come from the knowledge of the atomic predicate structure for future work.

Acknowledgments

The authors would like to thank Ivana Podnar, Martin Rajman, Wojciech Galuba and other LSIR group members for their valuable feedback.

7. REFERENCES

[1] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM'04, Portland, USA*, 2004.

[2] B. Bhattacharjee, S. Chawathe, V. Gopalakrishnan, P. Keleher, and B. Silaghi. Efficient peer-to-peer searches using result-caching. In *IPTPS'03, Berkeley, CA, USA*, 2003.

[3] M. Cai, M. Frank, J. Chen, and P. Szekely. Maan: A multi-attribute addressable network for grid information services. *Journal of Grid Computing*, 2(1), 2004.

[4] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009, 2001.

[5] P. Cudré-Mauroux and K. Aberer. A decentralized architecture for adaptive media dissemination. In *ICME'02, Lausanne, Switzerland*, 2002.

[6] A. Datta, M. Hauswirth, R. Schmidt, R. John, and K. Aberer. Range queries in trie-structured overlays. In *P2P'05, Konstanz, Germany*, 2005.

[7] L. Garcés-Erice, P. Felber, E. W. Biersack, G. Urvoy-Keller, and K. W. Ross. Data indexing in peer-to-peer dht networks. In *ICDCS'04, Hachioji, Tokyo, Japan*, 2004.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA, 1979.

[9] A. Kothari, D. Agrawal, A. Gupta, and S. Suri. Range addressable network: A p2p cache architecture for data ranges. In *P2P'03, Linköping, Sweden*, 2003.

[10] B. T. Loo, R. Huebsch, J. M. Hellerstein, S. Shenker, and I. Stoica. Enhancing p2p file-sharing with an internet-scale query processor. In *VLDB'04, Toronto, Canada*, 2004.

[11] J. Lu and J. Callan. Federated search of text-based digital libraries in hierarchical peer-to-peer networks. In *ECIR'05, Santiago de Compostela, Spain*, 2005.

[12] B. Mandhani and D. Suciu. Query caching and view selection for xml databases. In *VLDB'05, Trondheim, Norway*, 2005.

[13] I. Podnar, T. Luu, M. Rajman, F. Klemm, and K. Aberer. A peer-to-peer architecture for information retrieval across digital library collections. In *ECDL'06, Alicante, Spain (to appear)*, 2006.

[14] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[15] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Middleware'03, Rio de Janeiro, Brazil*, 2003.

[16] O. D. Sahin, A. Gupta, D. Agrawal, and A. E. Abbadi. A peer-to-peer framework for caching range queries. In *ICDE'04, Boston, USA*, 2004.

[17] G. Skobeltsyn, M. Hauswirth, and K. Aberer. Efficient processing of XPath queries with structured overlay networks. In *ODBASE'05, Agia Napa, Cyprus*, 2005.

[18] T. Suel, C. Mathur, J.-W. 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 *WebDB'03, San Diego, California*, 2003.

[19] C. Tang and S. Dwarkadas. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *NSDI'04, San Francisco, CA, USA*, 2004.

[20] C. Tryfonopoulos, S. Idreos, and M. Koubarakis. Publish/subscribe functionalities for future digital libraries using structured overlay networks. In *DELOS'05, Schloss Dagstuhl, Germany*, 2005.

[21] http://en.wikipedia.org.

[22] http://download.wikimedia.org/enwiki/20060518.

[23] M. R. Yong Yang, Rocky Dunlap and B. F. Cooper. Performance of full text search in structured and unstructured peer-to-peer systems. In *INFOCOM'06, Barcelona, Spain*, 2006.

[24] J. Zhang and T. Suel. Efficient query evaluation on large textual collections in a peer-to-peer environment. In *P2P'05, Konstanz, Germany*, 2005.