Search Result Caching in Peer-to-Peer Information Retrieval Networks

Almer S. Tigelaar, Djoerd Hiemstra and Dolf Trieschnigg

University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands
{tigelaaras, hiemstra, trieschn}@cs.utwente.nl

Abstract For peer-to-peer web search engines it is important to quickly process queries and return search results. How to keep the perceived latency low is an open challenge. In this paper we explore the solution potential of search result caching in large-scale peer-to-peer information retrieval networks by simulating such networks with increasing levels of realism. We find that a small bounded cache offers performance comparable to an unbounded cache. Furthermore, we explore partially centralised and fully distributed scenarios, and find that in the most realistic distributed case caching can reduce the query load by thirty-three percent. With optimisations this can be boosted to nearly seventy percent.

1 Introduction

In peer-to-peer information retrieval a network of peers provide a search service collaboratively. We define a peer as a computer system connected to the Internet. The term peer refers to the fact that in a peer-to-peer system all peers are considered equal and can both supply and consume resources. In a peer-to-peer network each additional peer adds extra processing capacity and bandwidth in contrast with typical client/server search systems where each additional client puts extra strain on the server. When such a peer-to-peer network has good load balancing properties it can scale up to handle millions of simultaneous peers. However, the performance is strongly affected by how well it can deal with the continuous rapid joining and departing of peers which is called *churn*.

We study peer-to-peer information retrieval systems where each peer contains, and maintains an index over, a subset of all the documents in the system. Since presumably relevant search results can be located at any peer it is difficult to route a query to the right one. This problem is commonly approached using different network topologies and replication of index data [1, 2, 3].

In this paper we explore search result caching. We assume that for each query there is a peer that can provide a set of original search results. If this query is posed often that peer would cripple under the demand for providing this set over and over again. Hence, we propose that each peer that obtains search results for a particular query *caches those results*. The effect is that search results for popular queries can be obtained from many peers: *high availability*, and the load on the peer that provided the original results is reduced: *load balancing*.

We define the following research questions:

1. What fraction of queries can be potentially answered from caches?
2. How can the cache hit distribution be characterised?
3. What is the distribution of cached result sets given an unbounded cache?
4. What is the effect of bounding the cache: how does the bound and cache policy affect performance?
5. What optimisations can be applied to make caching more effective?
6. How does churn affect caching?

Most research in peer-to-peer information retrieval focuses on simulating networks of hundreds [4] to thousands [3] of peers. In contrast, our experiments are of a larger scale: using over half a million peers. To our knowledge, there is no previous scientific work that investigates the properties of networks of this size. Our motivation is that large peer-to-peer information retrieval networks deserve more attention because of their real-world potential [5], and that this size is in the range of operational peer-to-peer networks used for other applications [6].

This paper is organised as follows: we discuss related work in Section 2. We explain our experiment set-up in Section 3 and show the results of experiments in sections 4 and 5. Finally, Section 6 concludes the paper.

2 Related Work

Markatos [7] analysed the effectiveness of caching search results for a centralised web search engine combined with a caching web accelerator. His experiments suggest that one out of three queries submitted has already been submitted previously. Cache hit ratios between 25 to 75 percent are possible. He showed that even a small bounded cache (100MB) can be effective, but that the hit ratios still increase slowly when the cache size is increased to several gigabytes. The larger the cache, the less difference the policy for replacing items in the cache makes. He recommends taking into account both access frequency and recency.

Skobeltsyn and Aberer [4] investigated how search result caching can be used in a peer-to-peer information retrieval network. When a peer issues a query it first looks in a distributed meta-index, kept in a distributed hash table, to see if there are peers with cached results for this query. If so, the results are obtained from one of those peers, but if no cached results exist, the query is broadcast through the entire network. The costs of this fallback are $O(n)$ for a network of n peers. In our experiments we do not distribute the meta-index, but focus only on the distributed cache. An additional difference is that they always use query subsumption: obtaining search results for subsets of the terms of the full query. They claim that with subsumption cache hit rates of 98 percent are possible as opposed to 82 percent without. The authors also utilized bounded caches, but do not show the effect of different limits.

Bhattacharjee et al. [8] propose using a special data structure combined with a distributed hash table to efficiently locate cached search result sets stored for particular term intersections. This is particularly helpful in approaches that

store an inverted index with query terms as it reduces the amount of network traffic necessary for performing list intersections for whole queries. This could be considered to be bottom-up caching: storing results for individual terms, then combinations of terms up to the whole query level. Whereas subsumption is top-down caching: storing results for the whole query, then for combinations of terms and finally for individual terms.

3 Experiment Set-up

3.1 Introduction

Our experiments intend to give insight into the *maximum benefits* of caching. Each experiment has been repeated at least five times, averages are reported, no differences were observed that exceeded 0.5 percent. We assume that there are three types of peers: *suppliers* that have their own locally searchable index, *consumers* that have queries to issue to the network, and *mixed peers* that have both. In our experiments the indices themselves do not actually exist and we assume that for each query a fixed set of pre-merged search results is available. We also assume that all peers cooperate in caching search result sets.

3.2 Collection

To simulate a network of peers posing queries we use a large search engine query log [9]. This log consists of over twenty million queries of users recorded over a time span of three months. Each unique user in the log is a distinct peer in our experiment for a total of 651,647 peers. We made several adjustments. Firstly, some queries are censored and appear in the log as a single dash [10]: these were removed. Secondly, we removed entries by one user in the log that poses an unusually high number of queries: likely some type of proxy. Furthermore, we assume that a search session lasts at most one hour. If the exact same query was recorded multiple times in this time window, these are assumed to be requests for subsequent search result pages and are used only once in the simulation. Table 1 shows statistics regarding the log. We play back the log in chronological order. One day in the log, May 17th 2006, is truncated and does not contain data for the full day. This has consequences for one of our experiments described later. For clarity: we do not use real search results for the queries in the log. In our experiments we make the assumption that specific subsets of peers have search result sets and obtain experimental results by counting hits only.

Table 1. Query log statistics.

Users	651,647
Queries (All)	21,082,980
Queries (Unique)	10,092,307

3.3 Tracker

For query routing we introduce the *tracker* that keeps track of which peers cache search result sets for each query. This is inspired by BitTorrent [11]. However, in BitTorrent the tracker is used for locating a specific file: *exact search*. A hash sequence based on a file's contents yields a list of all peers that have an exact copy of that particular file. In contrast, we want to obtain a list of peers which have cached search result sets for a specific free-text query: *approximate search*.

The tracker can be implemented in various ways: as a single dedicated machine, as a group of high capacity machines, as a distributed hash table or by fully replicating a global data index over all peers. Let us first explore if a single machine solution is feasible. The tracker needs to store only queries and mappings to all peers in the network. We can make a rudimentary calculation based on our log: storing IPv6 addresses for all the 650,000 peers would take about 10MB. Storing all the queries in the log, assuming an average query length of 15 Bytes [9, 12], would take about 315 MB. Even including the overhead of data structures we could store this within 1GB. Consider that most desktop machines nowadays have 4GB of main memory and disk space in the range of TeraBytes. However, storage space is not the only aspect to consider, bandwidth is equally important. Assume that the tracker is connected to a 100 Megabit line, which can transfer 12.5 MB per second. The tracker receives queries, 15 Bytes each, and sends out sets of peer addresses, let us say 10 per query: 160 Bytes. This means that a single machine can process 81,920 queries per second. This would work even if 12 percent of the participating peers would query it every second.

In our calculation we have made many idealizations, but it shows that a single machine can support a large peer-to-peer network. Nevertheless, there are three reasons to distribute the tracker. Firstly, a single machine is also a single point of failure: if it becomes unreachable, due to technical malfunction or attacks, the peer-to-peer network is rendered useless. Secondly, a single machine may become a bottleneck even outside its own wrongdoing: for example due to poor bandwidth connections of participating peers. Thirdly, putting all this information in one place opens up possibilities for manipulation.

4 Centralised Experiments

Let us first consider the case where one supplier peer in the system is the only peer that can provide search results. This peer does not pose queries. This scenario provides a baseline which resembles a centralised search system. Calculating the query load is trivial in this case: all 21 million queries *have to be* answered by this single central supplier peer. However, what if the search results provided by the central supplier peer can be cached by the consuming peers? In this scenario the tracker makes the assumption that all queries are initially answered by the central peer. When a consuming peer asks the tracker for advice for a particular query, this peer is registered at the tracker as caching search results for that query. Subsequent requests for that same query are offloaded to caching

peers by the tracker. When there are multiple caching peers for a query, one is selected randomly. Furthermore, we assume unbounded caches for now.

Figure 1 shows the number of search results provided by the origin central supplier peer and the summed number of hits on the caches at the consumer peers. It turns out that results for about half of the queries need to be given by the supplier at least once. The other half can be served from the caches of other peers. Caching can reduce the load on the central peer by about 50 percent. This suggests that about half the queries we see are unique. Skobeltsyn and Aberer [4] find that only 18 percent of the queries they use are unique. Perhaps this is because their log is a Wikipedia trace as this is inconsistent with our findings and contradicts observations of large web search engines [13, p. 183]

Caching becomes more effective as more queries flow through the system. This is due to the effect that there are increasingly more repeated queries and less unique queries. So, you always see slightly fewer new queries than queries you have already seen as the number of queries increases. Perhaps there is mild influence of Heap's law at the query level [13, p. 83].

How many results can a peer serve from its local cache and for how many does it have to consult caches at other peers? The local cache hit ratio climbs from around 22 percent for several thousand queries to 39 percent for all queries. These local hits are a result of re-search behaviour [14]. The majority of cache hits, between 61 and 78 percent, is on external peers.

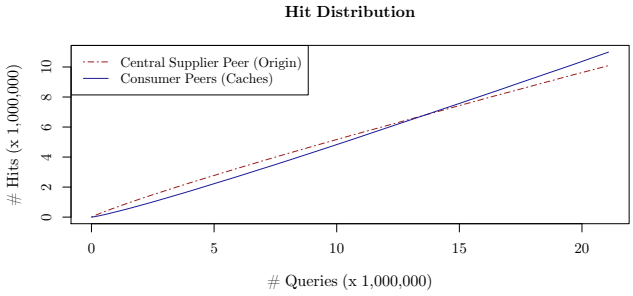

Figure 1. Distribution of hits when peers perform result caching.

Let us take a closer look at external hits. We define a peer's share ratio as:

$$shareratio = \#cachehits/\#queries \tag{1}$$

where *cachehits* is the number of external hits on a peer's cache: all cache hits that are not queries posed by the peer itself. *Queries* is the number of queries issued by the peer. A *shareratio* of 0 means that a peer's cache is never used for answering external queries, 1 that a peer answers as many queries as it poses, and above 1 indicates that a peer serves results for more queries than it sends.

Figure 2 shows that about 20 percent of peers does not share anything at all. It turns out that the majority of peers, 68 percent, at least serve results for some queries, whereas only 12 percent, about 80,000 peers, serve results for more queries than they issue.

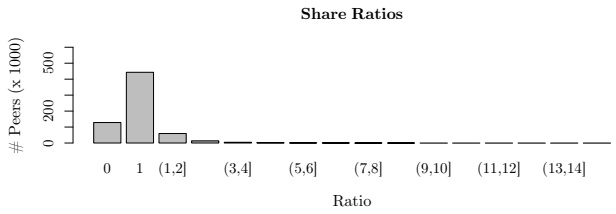

Figure 2. Observed share ratios.

4.1 Required Cache Sizes

So far we have assumed caches of unbounded size. This is not very realistic since machines in a peer-to-peer network have limited resources. Let us try to find out how big a cache we really need. Figure 3 shows the distribution of the number of cached items per peer for the previous experiment. We see that the vast majority of peers, about 225 000, cache between 1 and 5 search result sets. The graph is cut-off after 250 results, but extends to the highest number of cached items seen at a single peer: about 7500.

How much space does it take to store a set of search results? Assume that each set of results consists of 10 items and that each item consists of a URI, a summary and some additional meta-data, taking up 1K of space: 10K per set. Even for the peer with the largest number of cached results this takes only 73MB. However, a cache of 5 items, 50KB, is much more typical. Table 2 gives an overview of storage requirements for various search result set sizes. Most modern personal computers can keep the entire cache in main memory, even with a supporting data structure like a hash table.

Table 2. Cache storage requirements in MegaBytes (MB). Assumes each search result takes up 1KB: 5 results for low, 100 for medium and 7500 for high.

Result Set Size	Low (5)	Medium (100)	High (7500)
10	0.05	1	73
100	0.5	10	730
1000	5	98	7300

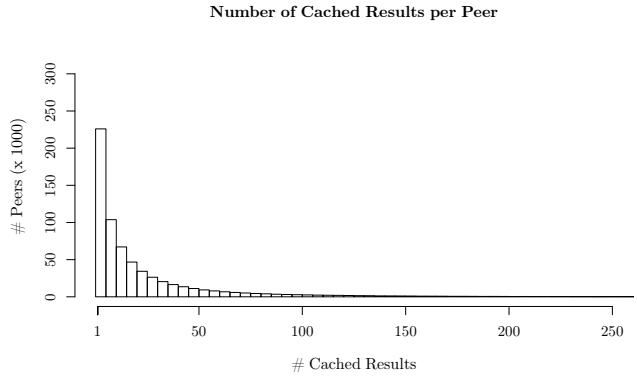

Figure 3. Observed cache sizes. Each bar represents 5 search results. The horizontal axis extends to 7500. The visible part of the graph covers 99.2 percent of all peers, each peer caches at least one search result.

4.2 Bounded Caches

As suggested in the previous section: it is possible to use unbounded caches for at least some time. However, it is not very desirable to do so for two reasons. Firstly, if systems run for an extended period of time, the cache has to be bounded somehow since it will run out of space eventually. Secondly, there is no point in keeping around result sets that are not requested any more.

We want to limit the size of the cache at some maximum number of search result sets to keep. To this end we investigate three different cache policies, with different limits on the cache size. When a new result set has to be inserted in the cache and the cache limit is reached the cache policy comes into play.

The most basic policy when the cache limit is reached is to throw out a random result set, this is called *Random Replacement* (RR) [15]. The advantage of this method is that it requires no additional administration at all. The downside is that we may be throwing away valuable sets from the cache. What is valuable is conventionally expressed using either frequency or recency which provides the motivation for the two other policies tested [7]. In the *Least Frequently Used* (LFU) policy the search result set which was consulted the least amount of times, meaning: which has the least hits, is removed. In *Least Recently Used* (LRU) the search result set that was least recently consulted is removed. In the case of LFU there can be multiple 'least' sets which have the same lowest hit count. If this occurs a random choice is made among those sets.

Figure 4 shows the hit distribution for the baseline unbounded cache and the RR, LFU and LRU caching strategies with various cache limits after running through the entire log. Experiments were conducted with per-peer cache limits of 5, 10, 20, 50 and 100 result sets. We can see that a higher cache limit brings the results closer to the unbounded baseline, which is what we would expect.

The most basic policy, Random Removal, performs worst particularly when the cache size is small (L5, L10). However, it performs almost the same as the LFU algorithm for large caches (L50, L100). In fact LFU performs quite poorly across the board. We believe this is caused by the fact that there can be many sets with the same hit count in a cache which degrades LFU to RR. For all cases the LRU policy is clearly superior. Although, the higher the limit, the less it matters what policy is used, also found by [7]. L100/LRU with 10 results per set takes only 1MB of space and achieves 99.1 percent of the performance of unbounded caches.

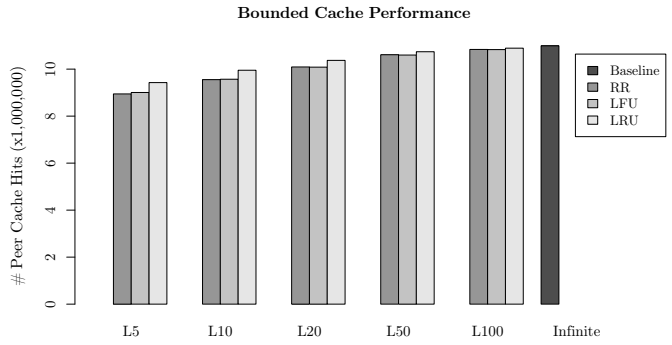

Figure 4. Bounded cache performance. The total number of queries is 21,082,980. The bars show the amount serviceable from peer caches for various per-peer cache size limits (5, 10, 20, 50 and 100) and strategies (RR, LFU and LRU). The rightmost bar shows the performance with unbounded caches.

4.3 Optimisations

In this section we use unbounded caches and investigate the impact of several optimisations: *stopword removal*, *reordering*, *stemming* and *query subsumption*. These techniques map multiple queries that were previously considered distinct to one common query representation. Since the number of representations is lower than the original number of queries, the strain of serving original search results on the central supplier peer is also lower. This capitalizes on the fact that there are cached copies of search result sets around for similar queries.

For stopword removal we remove words from queries that match those in a stopword list used by Apache Lucene consisting of 33 English terms. For reordering, the words in the query are alphabetically sorted, for example: from "with MacCutcheon you live happily ever after" to "after ever happily live MacCutcheon with you". The last common technique is stemming, for example from "airplane" and "airplanes" to "airplan". This example also shows the well known

drawback of stemming: that of reducing unrelated distinct meanings to the same form. We used the Porter2 English stemming algorithm [16].

We ran experiments with the three described techniques individually and all three combined. The first five rows of Table 3 show the results. We see that without any optimisations the central peer has to serve 47.9 percent of all queries. Applying stopping or re-ordering only marginally improves this by about half a percent. Stemming offers the best improvement: over 1.6 percent. Combining the techniques is quite effective and yields a 3.1 percent improvement in total, which is more than the sum of the individual techniques.

One final technique that is less commonly used is query subsumption [4]. When a full query yields no search results, subsumption breaks the query into multiple subqueries. This process iterates with increasingly smaller subqueries until at least one of these queries yields search results. The subqueries generated are combinations, with no repetition, of the terms in the full query. The length goes down each iteration, starting from $\text{len}(query) - 1$ terms to a minimum of 1 term. For example, given a resultless query q of length three: "A B C", we next try the three combinations of length two: "A B", "A C" and "B C". If that yields no results we try all combinations of length one, which are the individual terms "A", "B" and "C". The rationale for iterating top-down, from the whole query to the individual terms, is that longer queries are more specific and are thus expected to yield more specific, higher quality, results. Long queries generate an unwieldy number of possible subqueries. Therefore, we restrict the maximum number of generated combinations at any level to 1000.

In our experiment we evaluate at each iteration whether there is a query that yields at least one search result set. If so: all queries at that same iteration level for which there are cached result sets generate cache hits. Hence, for the example above, if for the full query "A B C" search results are not available, but there is at least one result at the level of individual terms: "A", "B" and "C". The full query can generate 1–3 cache hits: one for each individual term for which a result set is available. This thus causes the total amount of cache hits to increase beyond the number of original queries and simulates the effect of increased query load for merging result sets from multiple peers.

Table 3 shows the results. As mentioned the total amount of cache hits is different: 24 million for subsumption alone, a 13.6 percent increase. Nevertheless, performance improves with 21.4 percent less strain on the central peer. Combining subsumption with the three other techniques further increases the query total to nearly 26 million, but also further decreases the central peer load by 4.2 percent. The trade-off with subsumption is a higher total query load, but a lower load on the central peer. It reduces query-level caching to term-level caching which is known to have higher hit rates [13, p. 183]

All the discussed optimisations decrease precision in favour of higher recall. Hence, the quantity of search results for a particular query goes up, but the quality is likely to go down. Whether such a trade-off is justified depends on how sparse the query space is to begin with. However, for a general search engine, it certainly makes sense to apply some, if not all, of these techniques.

Table 3. Cache hits for various optimizations (x 1,000). Shows what party answers what query as an absolute number and percentage. The first five rows have a total query count of 21 million. The sixth 24 and the seventh 26 million.

	Central		Internal		External	
Baseline	10,092	47.9%	4,237	20.1%	6,754	32.0%
Sto(P)	9,993	47.4%	4,265	20.2%	6,824	32.4%
(R)eorder	9,992	47.4%	4,274	20.3%	6,816	32.3%
(S)tem	9,768	46.3%	4,359	20.7%	6,955	33.0%
P+R+T	9,449	44.8%	4,462	21.2%	7,172	34.0%
S(U)bsumption	6,352	26.5%	7,239	30.2%	10,365	43.3%
P+R+T+U	5,773	22.3%	8,335	32.1%	11,834	45.6%

5 Decentralised Experiments

Now that we have shown the effectiveness of caching for offloading one central peer, we make the scenario more realistic. Instead of a central peer we introduce n peers that are *both* supplier and consumer. These mixed peers are chosen at random. They serve search results, pose queries and participate in caching. The remaining peers are merely consumers that can only cache results.

The central hits in the previous sections become hits per supplier in this scenario. We further assume unbounded caches and no optimisations to focus on the differences between the centralised and decentralised case. How does the distribution of search results affect the external cache hit ratios of the supplier peers? We examine two distribution cases:

Single Supplier For each query there is always only exactly one supplier with unique relevant search results.

Multiple Suppliers The number of supplier peers that have relevant search results for a query depends on the query popularity. There is always at least one supplier for a query, but the more popular a query the more suppliers there are (up to all n suppliers for very popular queries).

For simplicity we assume in both cases that there is only one set of search results per query. In the first case this set is present at exactly one supplier peer. However, the second case is more complicated: among the mixed peers we distribute the search results by considering each peer as a bin covering a range in the query frequency histogram. We assume that for each query there is at least one peer with relevant results. However, if a query is more frequent it can be answered by more peers. The most frequent queries can be served by *all* n suppliers. The distribution of search results is, like the queries themselves, *Zipf* over the suppliers. We believe that this is realistic, since popular queries on the Internet tend to have many search results as well. In this case the random choice is between a variable number m of n peers that supply search results for a given query. Thus, when the tracker receives a query for which there are multiple possible peers with results it chooses one randomly.

We performed two experiments to examine the influence on query load. The first is based on the single supplier case. The second is based on the multiple suppliers case. For multiple suppliers we first used the query log to determine the popularity of queries and then used this to generate the initial distribution of search results over the suppliers. This distribution is performed by randomly assigning the search results to a fraction of the suppliers depending on the query popularity. Since normally the query popularity can only be approximated, the results represent an ideal outcome. We used n = 10,000 supplier peers in a network of 651,647 peers in total (about 1.53 percent). This mimics the Internet with a small number of websites and a very large number of surfing clients.

Figure 5 and Table 4 show the results. The number of original search results provided by the suppliers is about five percent higher than in the central peer scenario. This is the combined effect of no explicit offloading of the supplier peers by the tracker, and participation of the suppliers in caching for other queries. In the second case there is slightly more load on the supplier peers than in the first case: 57 percent versus 55 percent. The hit distribution in Figure 5 is similar even though the underlying assumptions are different. About 87 percent of peers answer between 1000 and 1500 queries. A very small number of peers answers up to about five times that many queries. Differences are found near the low end, which seems somewhat more spread in the single supplier than in the multiple suppliers case. Nevertheless, all these differences are relatively small. The distribution follows a wave-like pattern with increasingly smaller peaks: near 1300, 2500, 3700 and 4900 (not shown). The cause of this is unknown.

Table 4. Original search results and cache hits (x 1,000). 10,000 peers are suppliers operating in mixed mode.

	Single	Multiple
Suppliers (origin)	11,599	12,111
Consumers internal (caches)	3,683	3,930
Consumers external (caches)	5,801	5,042

5.1 Churn

The experiments thus far have shown the maximum improvements that are attainable with caching. In this section we add one more level of realism: we no longer assume that peers are on-line infinitely. We base this experiment on the single supplier case from the previous section where the search results are uniformly distributed over the suppliers. The query log contains timestamps and we assume if a specific peer has not issued a query for some period of time, its session has ended and its cache is temporarily no longer available. If the same peer issues a query later, comes back on-line, its cache becomes available again. This simulates churn in a peer-to-peer network where peers join and depart from the

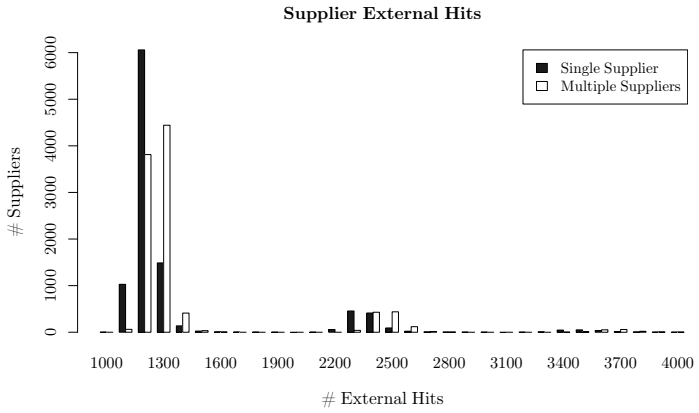

Figure 5. Supplier external hit distributions (n=10,000 suppliers).

network. We assume the presence of persistent peer identifiers, also used in real-world peer-to-peer systems [17]. All peers, including supplier peers, are subject to churn. For bootstrapping: if there are no suppliers on-line at all, an off-line one is randomly chosen to provide search results.

Assuming that all peers are on-line for a fixed amount of time is unrealistic. Stutzbach and Rejaie [6] show that download session lengths, post-download lingering time and the total up-time of peers in peer-to-peer file sharing networks are best modelled by using *Weibull distributions*. However, our scenario differs from file sharing. An information retrieval session does not end when a search result has been obtained, rather it spans multiple queries over some length of time. Even when a search session ends, the machine itself is usually not immediately turned off or disconnected from the Internet. This leads us to two important factors for estimating how long peers remain joined to the network. Firstly, there should be some reasonable minimum that covers at least a browsing session. Secondly, up-time should be used rather than 'download' session length. As soon as a peer issues its first query we calculate the remaining up-time of that peer in seconds as follows :

$$remaininguptime = 900 + (3600 \cdot 8) \cdot w \tag{2}$$

where w is a random number drawn from a Weibull distribution with $\lambda = 2$ and $k = 1$. The w parameter is usually near 0 and very rarely near 10. The up-time thus spans from at least 15 minutes to at most about 80 hours. About 20 percent of the peers is on-line for longer than one day. This mimics the distribution of up-times as reported in [6], making the assumption the uptime of peers in file sharing systems resembles that of information retrieval systems.

Figure 6 shows the results: the number of origin search results served by suppliers as well as the number of internal and external hits on the caches of

consumer peers. We see that the number of supplier hits increases to over 12.75 million: over 1.16 million more compared to the situation with no churn. The majority of this increase can be attributed to a decrease in the number of external cache hits. The dotted cloud shows the size of the peer-to-peer network on the right axis: this is the number of peers that is on-line simultaneously. We can see that this varies somewhere between about 30,000 and 80,000 peers. There is a dip in the graph caused by the earlier described log truncation.

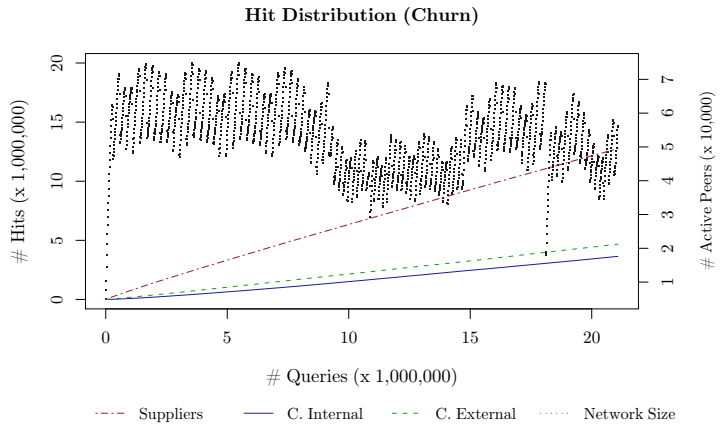

Figure 6. Distribution of hits under churn conditions (N=651,647 peers).

We also ran this distributed experiment with churn with a L100 LRU cache and all optimizations from the previous section enabled: stopping, re-ordering, stemming and query subsumption. This yields a cache hit ratio of 69 percent (for 25,45 million queries) for this most realistic scenario.

6 Conclusion

We conducted several experiments that simulate a large-scale peer-to-peer information retrieval network. Our research questions can be answered as follows:

1. At least 50 percent of the queries can be answered from search result caches in a centralised scenario. For the decentralized case cache hits up to 45 percent are possible.
2. Share ratios, the rate between cache hits and issued queries, are skewed which suggests that additional mechanisms are needed for cache load balancing.
3. The typical cache size is small, with outliers for eagerly consuming peers. Peers that issue a lot of queries also provide lots of cached results.

4. Small bounded caches approach the performance of unbounded caching. The Least Recently Used (LRU) cache replacement policy consistently outperforms the other policies. However, the larger the cache the less the policy matters. If each peer were to keep just 100 cached search result sets the performance is 99.1 percent of unbounded caches.
5. We have shown that stopword removal, stemming and alphabetical term re-ordering can be combined to boost the amount of cache hits by about 3.1 percent. Query subsumption can increase cache hits by 21.4 percent, to nearly 80 percent, but also imposes a higher total query load. All of the optimisation techniques trade search result quality for quantity. However, they all improve the effective usage of caches.
6. Introducing churn reduces the maximum attainable cache hits to 33 percent (-12 percent) without optimizations and 69 percent (-11 percent) with optimizations.

We have shown the potential of caching under increasingly realistic conditions using a single large query log. Caching search results significantly offloads the origin suppliers that provide search results under all considered scenarios using this log. These experiments could be extended by adding extra layers of realism. For example individual search results could be considered instead of fixed search results per query, allowing merging and construction of new search result sets.

We have explored several fundamental caching policies showing that Least Recently Used (LRU) is the best approach for our scenario. However, more advanced policies could be explored that include frequency as a component such as 2Q, LRFU or ARC [7, 18, 15]. These techniques combine advantages of LRU and LFU. Nevertheless, In reality there may be more than just LRU/LFU to take into account. For example queries pertaining to current events for which the relevant search results frequently change. The result sets for such queries should have a short time to live, whereas there are queries for which the search results change very rarely, they could be cached much longer. Making informed decisions about invalidation requires knowledge about the rate of change for particular queries [19]. Perhaps this information, or an estimate thereof, could be made an integral part of the search results, similar to the way in which Domain Name System (DNS) records work. Furthermore, we have assumed that the capacity of the tracker is unbounded. However, a policy similar to what is used to maintain peer caches could be applied there too. This does have the consequence that the tracker looses track of search results which are available, but for which the mappings have been thrown away. Finally, we have not investigated peer selection and result merging, both of which are relevant for real-world systems [5].

7 Acknowledgements

This paper was created using only Free and Open Source Software. We gratefully acknowledge the support of the Netherlands Organisation for Scientific Research (NWO) under project 639.022.809.

References

[1] Cuenca-Acuna, F.M., Martin, R.P., Nguyen, T.D.: Planetp: Using gossiping to build content addressable peer-to-peer information sharing communities. In: Proceedings of HPDC, Seattle, Washington, US (June 2003)

[2] Suel, T., Mathur, C., Wu, J.w., Zhang, J., Delis, A., Kharrazi, M., Long, X., Shanmugasundaram, K.: Odissea: A peer-to-peer architecture. In: Proceedings of WebDB, San Diego, CA, US (June 2003) 67–72

[3] Lu, J., Callan, J.: Full-text federated search of text-based digital libraries in peer-to-peer networks. Information Retrieval **9**(4) (2006) 477–498

[4] Skobeltsyn, G., Aberer, K.: Distributed cache table: efficient query-driven processing of multi-term queries in p2p networks. In: Proceedings of P2PIR, Arlington, Virginia, US (November 2006) 33–40

[5] Lu, J.: Full-Text Federated Search in Peer-to-Peer Networks. PhD thesis, Carnegie Mellon University (2007)

[6] Stutzbach, D., Rejaie, R.: Understanding churn in peer-to-peer networks. In: Proceedings of IMC, Rio de Janeiro, BR (October 2006) 189–202

[7] Markatos, E.P.: On caching search engine query results. Computer Communications **24**(2) (February 2001) 137–143

[8] Bhattacharjee, B., Chawathe, S., Gopalakrishnan, V., Keleher, P., Silaghi, B.: Efficient peer-to-peer searches using result-caching. In: Proceedings of IPTPS, Berkely, CA, US (February 2003) 225–236

[9] Pass, G., Chowdhury, A., Torgeson, C.: A picture of search. In: Proceedings of InfoScale, Hong Kong (May 2006) 1

[10] Brenes, D.J., Gayo-Avello, D.: Stratified analysis of aol query log. Information Sciences **179**(12) (2009) 1844 – 1858

[11] Cohen, B.: Incentives build robustness in bittorrent. In: Proceedings of P2PEcon, Berkeley, CA, US (June 2003)

[12] McNamee, P., Mayfield, J.: Character n-gram tokenization for european language text retrieval. Information Retrieval **7**(1) (2004) 73–97

[13] Croft, W.B., Metzler, D., Strohman, T.: Search Engines: Information Retrieval in Practice. Pearson Education (2010)

[14] Teevan, J., Adar, E., Jones, R., Potts, M.A.S.: Information re-retrieval. In: Proceedings of SIGIR, Amsterdam, NL (July 2007) 151–158

[15] Podlipnig, S., Böszörmenyi, L.: A survey of web cache replacement strategies. ACM Computing Surveys **35**(4) (December 2003) 374–398

[16] Porter, M.F.: The english (porter2) stemming algorithm (2001) snowball.tartarus.org/algorithms/english/stemmer.html (January 2011).

[17] Pouwelse, J.A., Garbacki, P., Wang, J., Bakker, A., Yang, J., Iosup, A., Epema, D.H.J., Reinders, M., van Steen, M.R., Sips, H.J.: Tribler: A social-based peer-to-peer system. Concurrency and Computation: Practice and Experience **20**(2) (2008) 127–138

[18] Megiddo, N., Modha, D.S.: Arc: A self-tuning, low overhead replacement cache. In: Proceedings of FAST, Berkeley, CA, US (2003) 115–130

[19] Blanco, R., Bortnikov, E., Junqueira, F., Lempel, R., Telloli, L., Zaragoza, H.: Caching search engine results over incremental indices. In: Proceedings of SIGIR, Geneva, CH (July 2010) 82–89