Distributed Popularity Indices

Ashish Gupta Peter Dinda Fabian Bustamante
{ashish,pdinda,fabianb}@cs.northwestern.edu
Department of Computer Science, Northwestern University

1. INTRODUCTION

Distributed hash tables (DHTs) are a distributed, peer-to-peer analogue of hash indices in database systems. Given a key, a DHT returns a pointer to the associated object. DHTs have also be extended to support "keyword" queries [5, 2] (object identified by multiple keys). Fundamentally, however, these approaches all return a undirected sample of the full result set. Unfortunately, most applications are interested in the most *popular* members of the result set. In other words, if all the objects in the result set were to be ranked in descending order of the number of accesses to the object in a given time interval, the application's interest decreases the further down the ranked list it goes.

We are developing distributed popularity indices (DPIs). Suppose a DHT supports two query primitives. The first simply finds an object given a key:

$$\mathit{Lookup} : k \rightarrow d$$

while the second provides keyword queries:

$$\mathit{Query} : \{w_1, w_2, \ldots\} \rightarrow \{k_1, k_2, \ldots\}$$

where the w_i are keywords and the k_i are the keys of the objects that have all of those keywords associated with them. A DPI supports queries of the form

$$\mathit{LookupPop} : k \rightarrow (d, p)$$

where p is the popularity of the object associated with key k, and the conjunctive query

$$\mathit{QueryPop} : (\{w_1, w_2, \ldots\}, n) \rightarrow \{k_1, k_2, \ldots, k_n\}$$

which is similar to *Query* except that the keys of the n most popular objects are returned. As with a DHT, a DPI also must support *Insert*, *Update*, and *Delete* primitives. There is an additional primitive

$$\mathit{Visit} : (k, w_1, w_2, \ldots, v) \rightarrow .$$

that indicates that object associated with k (and its associated keywords w_i) has been visited v times.

While the DPI is a distributed structure that we either generate on the fly in response to query or maintain persistently within a DHT, copies of at least portions of it can be cached locally on the client.

Gupta is a Ph.D. student. Dinda and Bustamante are faculty.

Applications

Beyond the obvious application in peer-to-peer file sharing communities, DPIs have many uses. Consider replication. Suppose that the system replicates popular objects up to z times by using z different hash functions. A client would merely issue a *LookupPop* to determine how many of the hash functions can be used for a particular key, providing a simple decoupling.

Consider web search. Arguably, link structures, as used in PageRank, and aggregated bookmarks as in social bookmarking, are proxies for the extent to which a page has been visited. Web clients could push *Visit*s into the DPI as pages were visited. A *QueryPop* would then be able to extract the top n most visited pages associated with a set of keywords. If the DPI automatically reduced accumulated popularity of objects over time by an exponential response with some time constant, we would know what pages were significant in the context of some set of keywords for a window of time ending in the present: a zeitgeist query.

2. EXPLOITING REVERSIBLE SKETCHES

A k-ary sketch [3] is a variant of a Bloom filter [4] that captures popularity in a highly condensed form. A key is inserted into a Bloom filter by updating m hash tables of size s/m, each fronted with a different uniform hash function. Each bucket contains a bit and the update is a bit-or of each bucket determined by the key with one. In a sketch, the buckets contain integers and the update involves incrementing the buckets determined by the key. A Bloom filter provides a constant size representation of a potentially very large set of keys, with the caveat that it is imperfect—spurious keys may also be included. Similarly, a sketch is a constant size, but imperfect representation of an ideal popularity index that would map from key to number of visits. A key may appear more popular than it really is, but this noise affects *unpopular* keys much more than popular ones. A sketch, when queried with a key, provides an estimate of the number of updates (visits) to that key whose accuracy increases with the number of visits.

Surprisingly, given that sketches are based on hashes, there exists a *reverse-hashing* formulation that is *reversible* for popular keys, meaning that it is possible to determine the most visited keys from the sketch, as well as the number of updates to them, without knowing the keys in advance [6]. Our *Visit* maps to a sketch update, while the popularity p from *LookupPop* is determined by using a sketch S_{all} in the forward direction. S_{all} is computed over all keys. *QueryPop* is implemented using the per-keyword sketches $S_{w_1}, S_{w_2}, \ldots$ in the reverse direction and aggregating the results.

We can generate DPIs for use for individual queries, and DPIs that persist. We refer to these approaches as query-driven and update-driven indices.

3. QUERY-DRIVEN INDICES

Query-driven indices are computed and aggregated on the fly in response to *LookupPop*s and *QueryPop*s. For *LookupPop*, the client simply finds the node associated with k and asks it for the popularity of k, which the node stores as a simple counter associated with the object. *Visit* simply increments this counter.

For *QueryPop*, we use a DHT that provides keyword search, such as Magnolia [2, 1] or others [5]. We use *Query* to determine the set of keys of interest, and then contact each node associated with these keys. Each node constructs a sketch that reflects the counters associated with the matching keys it holds. We then sum all of these sketches and deliver them to the client. Notice that sketch summation is associative (indeed commutative), and so we can use a reduction tree to do this in log time. Recall also that the sketches are of fixed size. The client receives the summed sketch, which it reverses to determine the n most popular keys.

The final and intermediate sketches in the reduction tree could be cached in the DHT, associated with the keywords used and a special "__sketch" keyword, and a timeout. In the common case, even the reduction tree could be avoided for most queries.

4. UPDATE-DRIVEN INDICES

Update-driven indices have long-term persistence and make the workload involved with the popularity query independent of the workload for fetching objects. The key idea is that we distribute each of $S_{all}, S_{w_1}, S_{w_2}, \ldots$ using the DHT. We place the bucket (i, j) of sketch S_* into the DHT using the key "__sketch_*_i_j".

Visit is now more complex as it needs ultimately to update a bucket (actually m buckets in parallel) that could be anywhere in the DHT. However, notice that *Visit*, which simply increments the bucket, is an associative and commutative operation. Thus, when a node sees two *Visit*s being routed through it, it can simply sum their v arguments and emit a single *Visit*. Furthermore, if we permit the propagation of *Visit*s to be delayed (by giving each a deadline for when it must reach its bucket), the number of *Visit*s that we aggregate as we route them increases quickly.

LookupPop is straightforward—the client merely needs to query for the s buckets of S_{all} and then estimate the popularity from the reconstructed sketch. Of course, that sketch could also be cached locally and/or in the DHT.

QueryPop is a complex operation in an update-driven index. The initial step is to reconstruct the sketches $\{S_w : w \in querykeywords\}$. Unfortunately, it is not the case that if we sum these sketches we have the sketch that would have arisen if we had been managing a sketch for the query's conjunction of keywords. The most popular object for (w_1, w_2) may be very unpopular for w_1 and w_2 individually. It remains to be seen whether this loss of information is significant in practice for common workloads.

We expect that a better way to answer the *QueryPop* query is to compute correlations of *tracks* (sequences of hash buckets) through pairs of sketches. The most highly correlated tracks (highest covariance) are most likely associated with the keys that are most popular for the combination of keywords. However, naively, $m \times s/m$ sketches would result in $(s/m)^m$ tracks to be considered. We are working on dynamic programming approaches to this problem.

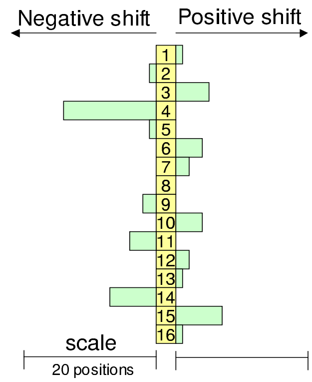

Figure 1: Accuracy of k-ary sketches for Zipf-distributed objects.

5. EXPERIMENTS

The effectiveness of our distributed popularity indices depends on the degree to which we can tolerate low accuracy, low precision answers for unpopular objects. Popularity must be strongly skewed so that the top n objects in terms of *Visit*s correspond to a large proportion of the visits even if n is small.

The popularity of documents and keywords tends to follow Zipf's rule, which is a rank power law, meaning that the proportion of the total visits in the system captured by the ith ranked document, is $\propto 1/i^{\alpha}$. Given such a distribution over keywords, how well do the top n documents recovered using reversible sketches capture the actual top n documents?

Figure 1 shows the results of a simple experiment to address this question. Here, we inserted over 20 million keys with their popularity assigned according the Zipf distribution with $\alpha = 1.0$. The most popular object was visited ~ 10000 times. Reverse-hashing algorithms were used to recover the top 20 keys from the sketch and then compared to ground truth. The figure shows the reported shift in rank of the top 20 keys. Clearly, the shifts are very small except for the 4th ranked item. The three keys that were not really in the top 20 were in the top 30.

The upshot of Figure 1 is that given keywords with a Zipf popularity rank distribution, a small sized sketch is quite capable of recovering the top n most popular keys with little error. This bodes well for the distributed popularity indices that we have described here.

6. REFERENCES

[1] GUPTA, A., SANGHI, M., DINDA, P., AND BUSTAMANTE, F. Magnolia: A novel dht architecture for keyword-based searching. *Technical Report, Northwestern University* (May 2005).

[2] GUPTA, A., SANGHI, M., DINDA, P., AND BUSTAMANTE, F. Magnolia: A novel dht architecture for keyword-based searching (poster). *NSDI 2005* (May 2005).

[3] KRISHNAMURTHY, B., SEN, S., ZHANG, Y., AND CHEN, Y. Sketch-based change detection: Methods, evaluation, and applications. In *Proc. of ACM SIGCOMM IMC* (2003).

[4] MITZENMACHER, M. Compressed bloom filters. *IEEE/ACM Transactions on Networking 10*, 5 (Oct. 2002), 604–612.

[5] REYNOLDS, P., AND VAHDAT, A. Efficient peer-to-peer keyword searching. In *Middleware* (2003), pp. 21–40.

[6] SCHWELLER, R., GUPTA, A., PARSONS, E., AND CHEN, Y. Reverse hashing for sketch-based one-pass change detection for high-speed networks. In *ACM SIGCOMM IMC* (2004).