Cache-aware load balancing of data center applications

Aaron Archer
Google
New York, New York
aarcher@google.com

Kevin Aydin
Google
New York, New York
kaydin@google.com

MohammadHossein Bateni
Google
New York, New York
bateni@google.com

Vahab Mirrokni
Google
New York, New York
mirrokni@google.com

Aaron Schild
UC Berkeley
Berkeley, California
aschild@berkeley.edu

Ray Yang
Google
New York, New York
rayy@google.com

ABSTRACT

Our deployment of cache-aware load balancing in the Google web search backend reduced cache misses by ∼0.5x, contributing to a double-digit percentage increase in the throughput of our serving clusters by relieving a bottleneck. This innovation has benefited all production workloads since 2015, serving billions of queries daily.

A load balancer forwards each query to one of several identical serving replicas. The replica pulls each term's postings list into RAM from flash, either locally or over the network. Flash bandwidth is a critical bottleneck, motivating an application-directed RAM cache on each replica. Sending the same term reliably to the same replica would increase the chance it hits cache, and avoid polluting the other replicas' caches. However, most queries contain multiple terms and we have to send the whole query to one replica, so it is not possible to achieve a perfect partitioning of terms to replicas.

We solve this via a voting scheme, whereby the load balancer conducts a weighted vote by the terms in each query, and sends the query to the winning replica. We develop a multi-stage scalable algorithm to learn these weights. We first construct a large-scale term-query graph from logs and apply a distributed balanced graph partitioning algorithm to cluster each term to a preferred replica. This yields a good but simplistic initial voting table, which we then iteratively refine via cache simulation to capture feedback effects.

PVLDB Reference Format:
Aaron Archer, Kevin Aydin, MohammadHossein Bateni, Vahab Mirrokni, Aaron Schild, Ray Yang, and Richard Zhuang. Cache-aware load balancing of data center applications. *PVLDB*, 12(6): 709-723, 2019.
DOI: https://doi.org/10.14778/3311880.3311887

Proceedings of the VLDB Endowment, Vol. 12, No. 6
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3311880.3311887

1. INTRODUCTION

Multiple commercial web search engines such as Baidu, Bing, Google, Yahoo, and Yandex now answer hundreds of millions to billions of queries per day [29], retrieving results from an index of many billions of documents. At this scale, containing hardware costs is vital, so teams of software engineers work for years to squeeze the maximum throughput out of each cluster. Caching is one key to the overall performance of the system. Although throughput is the focus of this paper, better caching improves both throughput and latency, so there is no tradeoff at play here.

We describe an innovation in the load balancing layer of the Google web search backend that has been applied to all production workloads since 2015. Our multi-phase approach (i) builds a predictive model of the caches, learning its parameters from massive logs of search queries, (ii) does so by employing a distributed balanced graph partitioning tool [6] that is designed to solve an NP-hard problem (heuristically) at scale, and (iii) iteratively optimizes the learned parameters of the predictive model with the aid of the query logs and a cache simulator. Our technique cut the miss rate of the relevant caches by a factor of roughly 0.5x, contributing to a double-digit percentage increase in the query throughput of our web search backend (details in Section 7).

Our work is built around the following central observation. In a distributed system that balances load across multiple identical servers (aka *replicas*), it is not always optimal to distribute the queries uniformly. If the replicas carry over mutable state between requests—such as by maintaining a data cache—then the decision of where to route a query affects both its own processing cost and that of succeeding queries, which infuses a complex feedback loop into the load balancing problem. This distinguishes our setting from most load balancing models, which assume the processing cost of a request depends only on the request and (possibly) the server that processes it, not on which other requests are sent to that server. Although we may have no control over the stream of requests seen by the load balancer, the balancer can exercise some control over the *stream of requests seen by each replica*. The more homogenous the request stream sent to a cache, the lower its miss rate tends to be. If our load balancer can cluster the incoming query stream such that the substream seen by each replica is more homogenous than the overall stream, then each replica should

incur a lower cache miss rate than it would if it saw either the full stream or a uniform random sample. Complicating matters, our system actually has many distributed load balancers running simultaneously, each handling a uniform random sample of the query stream. So we need a policy that allows the load balancers to consistently split their query streams into homogenous substreams, without communicating with each other. The main thrust of our paper is how to accomplish this, using a method we call *term-affinitized replica selection* (TARS).

An extreme version of this idea is *hash-based partitioning*, already present in the literature on distributed hash tables (Section 8), where requests are keys whose associated value must be looked up and returned. A distributed RAM cache is often used to minimize requests to bulk storage. Deterministic routing of the requests via a hash of the key ensures perfect affinitization between keys and cache replicas. Moreover, this method of affinitization allows multiple load balancers to coordinate without communicating.

However, this paper focuses on web search *retrieval*, which is the problem of querying an index to construct a list of all documents that match a given query. Retrieval is much harder to affinitize because each request contains *multiple* keys that must be looked up *together*. Each term has an associated *postings list* (PL), which is a list of the docids of all documents containing that term. In retrieval, the PL of each term in the query must be pulled into memory on the same server and then processed. Partitioning based on a hash of the query *as a whole* is ineffective for affinitizing retrieval because the vast majority of queries hitting our backend contain *multiple terms* and are unique within the period of time it takes to turn over the cache.

Our approach affinitizes *terms* rather than *queries*. For instance, all sports terms could be assigned to replica 0, and all clothing terms could be assigned to replica 1. But then which replica should we select for the query "tennis shoes"? Our TARS method surmounts this difficulty by instituting a *weighted vote among the terms in the query* (Section 2).

This paper is *not* about caching algorithms. TARS can be used in conjunction with *any* reasonable caching algorithm.

We now explain the relevant details of how a query is processed once it hits our web search backend. Barroso, Dean and Hölzle [7] described a search engine architecture that uses multiple clusters of servers scattered across the globe, and provides horizontal scalability within each of these search clusters via *doc-sharding* and *replication*. Our system shares these elements, which we describe here for completeness.

Two sets of jobs cooperate to process the query: *roots* (aka *query brokers*) and *leaves* (aka *index servers*). Most of the computation is done at the leaves. The set of all documents in the index and their PLs are partitioned into $\mathcal{S}$ pieces of nearly equal size, each of which is called a *shard* (aka *index shard* or *doc shard*). Each of these shards is replicated R times. Each leaf job serves one replica of one shard, so there are $R \cdot \mathcal{S}$ leaves total. A leaf may also be called a *serving replica*, *leaf replica*, or simply *replica*. When a query arrives at a root, a load balancer selects one replica of each shard, the root forwards the query to the $\mathcal{S}$ selected leaves, each of these performs retrieval on its index shard and returns results to the root, where the results are merged. Like the leaves, the roots are replicated to handle the load, but unlike the leaves, they are all identical.

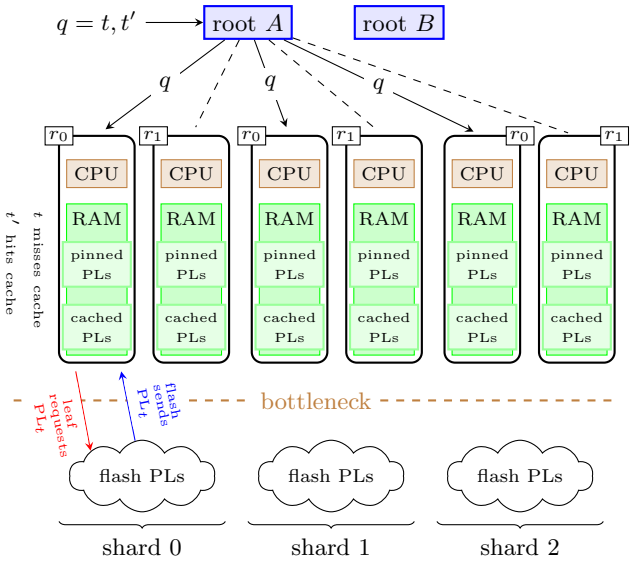

Figure 1: A replicated, doc-sharded search engine architecture, with two roots, six leaves, and $\mathcal{S} = 3, R = 2$. Here, root A fans out the query to replica 0 of each shard.

The aggregate size of all PLs in a shard is too much to fit in RAM on one leaf, so our system stores most of them on flash drives, pulling them into RAM when needed. The flash could be accessed remotely over the network (as in Figure 1), or mounted locally. The important aspect is that the bandwidth of the communication link between flash and RAM is limited, and is a critical bottleneck in our system. This motivates the use of an application-directed cache to hold these PLs in RAM on the leaf. The largest PLs are pinned permanently in RAM, to avoid unacceptable latency when pulling them into cache. See Figure 1.

Uniform sharding helps to balance the load across shards. To balance the load across the replicas of each shard, one's first impulse is to use either round-robin, uniform random selection, or deterministic selection based on query fingerprints.[1] As this paper will demonstrate, these approaches are suboptimal for caching. Selection by query fingerprint is the old method used by our system prior to this work, so it is our main baseline for comparison.

2. TARS: AN OVERVIEW

We now describe our load balancing system, TARS. A different copy of this load balancer runs independently within each root job. Whenever the root receives a new query to fan out to the leaves, the load balancer selects one replica of each shard. Since each root is identical and each shard is treated independently, we focus for the rest of the paper on a single root and *just the R replicas of a single shard*.

The overall goal of TARS is to cause the substream of queries received by each leaf to be as homogenous as possible (thereby decreasing the PL cache miss rate), while still balancing CPU utilization across the leaves. TARS consists of three components that work together to achieve this goal.

[1]Here, *fingerprint* means a hash function that achieves strong uniformity over its b-bit output range.

The first component is a table of *voting weights* w_t^r that we compute offline based on query logs, and load into memory on each root at startup. One should think of w_t^r as a vote by term t *against* being sent to replica r. This voting table is *static*—it does not change during serving. The second component is a set of dynamic *multipliers* μ^r, one for each replica r. The role of these multipliers is to allow TARS to shift load from one replica to another in real time in response to CPU load. The third component is a *voting rule* that combines the static voting table with the dynamic multipliers to select a replica. We describe all three components now, but the emphasis of this paper is on the first component: how to compute the table of voting weights.

We start by describing the voting rule. Since the terms pinned in RAM are never retrieved from flash, they do not affect the flash IO bottleneck depicted in Figure 1, so we exclude them from the vote. Therefore, we represent an incoming query q mathematically as the set of terms in the query whose PLs are stored on flash. When query q arrives at the root, TARS first computes, for each replica r, the sum of the voting weights over the terms in q:

$$v^r(q) = \sum_{t \in q} w_t^r. \tag{1}$$

As with the pinned terms, any terms missing from the voting table implicitly get zero weight. TARS then applies the replica-specific dynamic multipliers and takes the min. That is, it selects the replica

$$\arg\min_r v^r(q)\mu^r. \tag{2}$$

Thus, a high value of μ^r causes replica r to shed traffic. In the event of a tie, we fall back to selecting among the tied replicas using query fingerprints. The most common scenario causing a tie is when all terms in q are either pinned or missing from the table, making all votes zero.

Figure 2 depicts this process. Preferential voting has specialized the cache on replica 0 towards sports terms, and replica 1 towards clothing terms. Colors denote each term's preferred replica. Both "dress" and "shoes" want the clothing replica, so replica 1 wins that query, but "tennis" and "shoes" have a split preference. We imagine that "tennis" has a stronger preference for replica 0 than "shoes" does for replica 1, so replica 0 wins the query.

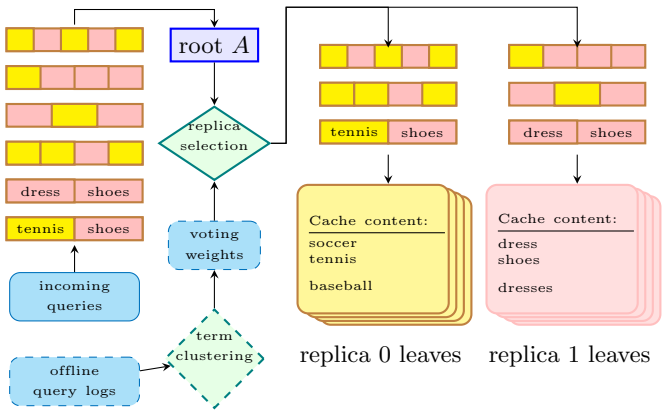

Figure 2: TARS in action.

We want w_t^r to represent the expected number of cache misses if we send term t to replica r. Then $v^r(q)$ is the total expected cache page misses if we send query q to replica r. Thus, TARS sends the query to the replica that minimizes the expected number of cache page misses, if all of the μ^r are equal. More generally, this lends a nice interpretation to the ratio $\mu^r/\mu^{\hat{r}}$ as being the maximum multiplicative increase in expected cache miss cost that we are willing to accept in order to shift a query away from replica r onto replica $\hat{r}$, so as to better balance the CPU load. Section 6 details how the μ^r are maintained. Since this aspect of TARS is less interesting than the voting weights, we ignore it from now until Section 6 (i.e., we assume $\mu^r = 1$ for each r). With this simplification, the TARS voting rule (2) boils down to selecting the replica

$$\arg\min_r v^r(q). \tag{3}$$

Most of this paper focuses on how to train the table of static voting weights w_t^r. Let PL_t denote the PL for term t, and let sz_t be the size of PL_t measured in cache pages, rounded up to the nearest integer (since a full page is the smallest block we read from flash). Suppose that p_t^r is our offline stationary probability estimate that term t is in cache on replica r. In order for w_t^r to represent the expected number of cache page misses, we should set

$$w_t^r = (1 - p_t^r)\text{sz}_t. \tag{4}$$

This formula assumes that each PL is pulled into and evicted from cache as a unit, which is true of our cache.

There is a circular dependency between the voting weights w_t^r and the cache hit probabilities p_t^r. If we change p_t^r, we want to update w_t^r via (4). But since the voting weights determine which substream of the root's query stream is seen by each replica, they determine the state of the caches and hence our estimates of p_t^r. Section 4 solves this problem, using an iterative method to converge to a fixed point of this feedback loop, using query logs and a cache simulator.

But first, we must generate an initial voting table. We do this using a distributed balanced graph partitioning tool that our team had previously developed [6]. We apply it to a large weighted bipartite graph between terms and queries, derived from query logs. We give the intuition behind our graph now, with full details in Section 3. It contains a node for each term and each query in the logs, connecting each term to each query containing it. Each edge has a *cost*, and each term node has a *mass* (each query node has zero mass). The costs and masses play very different roles. The edge cost between query q and term t represents the expected number of extra cache page misses incurred on PL_t if query q is sent to a replica other than the one preferred by term t. Meanwhile, the node mass on term t represents how many cache pages PL_t occupies on average (over time) in its preferred cache. We assign zero mass to the queries because they do not occupy space in the cache.

We then (approximately) solve an NP-hard balanced partitioning problem on this graph, aiming to minimize the total cost of edges cut, subject to balancing the term node mass across the R replicas. This constraint corresponds to the cache sizes being fixed and equal across all R replicas, while the edge cut corresponds to the *aggregate* page miss rate across all replicas. Each cluster in the partition identifies terms that tend to be queried together.

DEFINITION 1. *Given a partition of the set of all terms T into R clusters $T_0, \ldots, T_{R-1}$, the* binary voting table *associated with this clustering is the one given by equation* (4)*, using $p_t^r = 1$ if $t \in T_r$, and* 0 *otherwise.*

This voting table encodes the incredibly naive assumption that a term t always hits cache when it is sent to its preferred replica, and always misses cache otherwise. Nevertheless, even these initial voting weights turn out to be incredibly effective. Moreover, the edge cut cost of a clustering is a *very reliable proxy* for the cache miss rate induced by the resulting binary voting table, as shown by Figure 3. This plot shows 1000 different clusterings of the terms into five replicas. The point with the smallest miss rate (second from the left) corresponds to the clustering output by our balanced partitioner. Starting with this clustering, we randomly degrade it by varying amounts to create 999 other clusterings with various cut costs, and evaluate the cache miss rates of the corresponding binary voting tables. The plot shows a clear, near-linear relation between cut size in the graph and cache miss rate (details in Section 5.3).

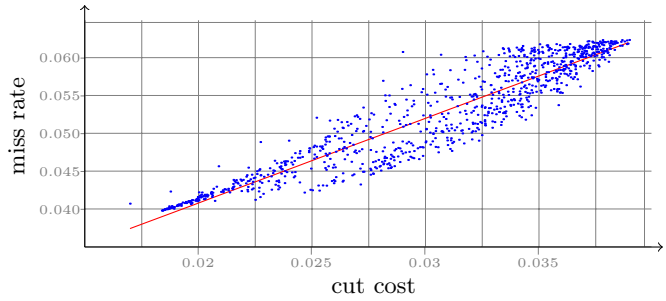

Figure 3: Cache miss rate vs. cut cost, as a fraction of total edge cost in term-query graph.

Balanced graph partitioning is not only NP-hard (Section 8), but also hard in practice, even on small graphs. The voting table we derive from the clustering improves substantially when we train on larger logs, and hence larger graphs. Thus, we require a distributed optimization tool to conquer the scale. Distributed graph computations are notoriously difficult. Even the most basic graph computations, such as connected components, become challenging for massive graphs, even if they are trivial on graphs that fit in memory (Section 8). The distributed balanced partitioning tool [6] that we previously built to handle massive graphs is therefore a key ingredient in the success of our overall approach.

There is one glaring problem with TARS as we have described it so far: although it should decrease the cache miss cost per query, there is no reason to believe it should balance the cache misses among the replicas. However, we argue in Section 9 that the aggregate miss rate is usually the right measure to address. This argument is easy when the flash is remote (as depicted in Figure 1), because the flash access cost is not borne directly by the leaf, so there is no need to balance it across leaves. The argument is more subtle when the flash is local to each replica, so we save that for Section 9. Moreover, Section 7 shows empirically that we reduce the tail of the leaf flash IO distribution even though we don't make an explicit effort to do so.

2.1 Our contributions

We evaluate the quality of a voting table by the aggregate cache miss rate it induces on all replicas. We measure this by running a query log through a cache simulator (or through a loadtest on a production cluster). We do not concern ourselves with the distribution of cache misses across replicas (but see Section 9). The set of all possible voting tables forms a complex landscape with many local minima. Our main technical contribution consists of several ways of exploring this landscape that have proven to be extraordinarily effective in practice (Table 3).

Our contributions include defining the problem of cache-aware load balancing when each request has multiple lookup keys, modeling it, and inventing TARS to solve it. Furthermore, our experience with TARS in our production system, re-confirmed by the detailed experimental study in Section 5, can be summarized as follows:

- TARS already beats the baseline fingerprint method (FPT) by a large margin, just by building a binary voting table (Definition 1) from a random clustering (RND). Thus, the TARS idea is already powerful even with a dumb voting table (Table 3).
- For binary voting tables derived from term clustering, the miss rate correlates strongly with the edge cut objective function. Thus, the cut objective and the quality of the optimization matter (Figure 3).
- Thus, using a balanced partitioning algorithm on the term-query graph of Section 3.2—built from real search query logs—would be expected to outperform RND. Indeed it does, by a substantial margin (Table 3).
- Our simulation-based iterative refinement method (Section 4) delivers substantial improvements, starting from either RND or BP. However, method BP still beats RND, even after refining both voting tables.
- Applying TARS to our production system reduced cache misses by a ∼0.5x factor, contributing to a double-digit percentage increase in query throughput.

We organize the rest of this paper as follows. Section 3 shows how to initialize our voting table using balanced partitioning on a term-query graph, and Section 4 explains how to iteratively refine it via cache simulation. Section 5 describes extensive simulations using real query logs from our production system. Section 6 describes how to maintain the dynamic load balancing multipliers μ^r and Section 7 reports results from our actual production system. Finally, Section 8 reviews related work, and Section 9 closes with a discussion of future research directions.

3. TERM CLUSTERING VIA BALANCED GRAPH PARTITIONING

This section uses clustering on a massive graph to generate a good table of term-replica voting weights w_t^r. First, we describe a general template for generating a voting table from any partitioning of the terms into R clusters. Next, we describe the *balanced graph partitioning* optimization problem, and how to use it to model our weight-setting problem. This part involves constructing a term-query graph from

query logs collected from our production system, then applying a balanced partitioning algorithm to cluster it. We then obtain a table of voting weights by applying the general template to this clustering.

Since the general clustering-to-table template can be applied to *any* clustering, it raises the question of whether the modeling in this section does any good. If we were to assign each term uniformly at random to a cluster and build a voting table out of that, would that table work just as well? Our experiments (Section 5) reveal the answer is no.

3.1 Voting tables from clustering

Let T denote the set of terms we wish to put in our table, and let $\{T_0, \ldots, T_{R-1}\}$ be a partition of T into R clusters. We represent this clustering by a function $C : T \to \{0, 1, \ldots, (R-1)\}$, where $C(t) = r$ iff $t \in T_r$. We want to build a voting table that will serve to affinitize term t to its preferred replica $C(t)$, as defined by the clustering.

Suppose we estimate *a priori* the probability of a cache hit as P_t when sending term t to its preferred replica $C(t)$, and as N_t when sending t to *any* of its non-preferred replicas. Tying this back to our notation from Section 2, this means

$$p_t^r = \begin{cases} P_t, & \text{if } C(t) = r \\ N_t, & \text{o.w.,} \end{cases} \tag{5}$$

so we set

$$w_t^r = \begin{cases} \mathrm{sz}_t(1 - P_t), & \text{if } C(t) = r \\ \mathrm{sz}_t(1 - N_t), & \text{o.w.} \end{cases} \tag{6}$$

This assumption seems ridiculous on its face, for at least two reasons. First, because of the chicken-and-egg problem that we raised in Section 2, where the voting table determines the hit rates while at the same time we desire to let the hit rates determine our table via equation (4). Second, because we have no reason to believe that there should be symmetry among the non-preferred replicas. In response, we point to the famous words of the statistician George Box: "All models are wrong but some are useful" [11]. In fact, we will obtain much mileage out of the simplistic assumption that $P_t = 1$ and $N_t = 0$ for all $t \in T$; but for the time being, let us assume only that $P_t \geq N_t$.

3.2 Graph definition & balanced partitioning

We now define a graph, based on a query log and the hit probabilities P_t, N_t assumed above for each term t. We then define the balanced partitioning problem and relate it to cache miss rates in our replica selection scheme. This will justify the way we defined the graph.

Let us assign a unique ID to each query in our log, and let I denote the collection of these IDs. Let q_i denote the query indexed by i, which we represent mathematically as a subset of the terms represented in our web search index. Let $T = \cup_{i \in I} q_i$ denote the set of all terms that appear in at least one query in the log. We construct a bipartite graph where the nodes on the left side are the terms in T, the nodes on the right side are the query IDs in I, and there is an edge (t, i) between query ID i and each of the terms $t \in q_i$. Any query appearing multiple times in the log will be represented a corresponding number of times on the right side of the graph. We assign a mass m_u to each node u and a cost c_e to each edge e as follows. We let $m_i = 0$ for all $i \in I$, and set $m_t := \mathrm{sz}_t(P_t - N_t)$ for each $t \in T$. For each edge e incident to t, we also set $c_e := \mathrm{sz}_t(P_t - N_t)$. Figure 4 depicts a small piece of an example graph of this type.

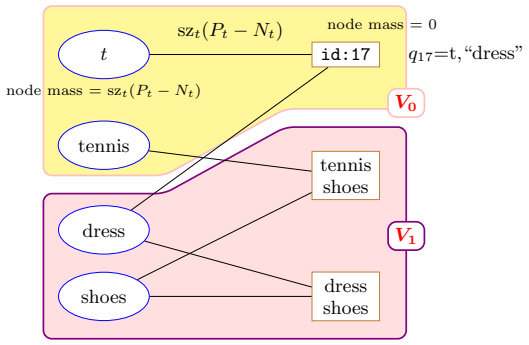

Figure 4: Example term-query graph, showing one node mass of each type and one edge cost. Terms T are on the left and query IDs I are on the right. A partition $\{V_0, V_1\}$ is also depicted.

Now consider *any* graph (V, E) with node mass m_u on each node $u \in V$ and edge cost c_e on each edge $e \in E$. We are also given a target number of clusters R and an imbalance tolerance $\varepsilon > 0$. Consider a partition $\mathcal{P}$ of the node set V into R clusters $\{V_0, \ldots, V_{R-1}\}$ such that the node mass is roughly balanced among the clusters, i.e., $m(V_j) := \sum_{u \in V_j} m_u \leq \frac{1+\epsilon}{R} \sum_{u \in V} m_u =: \frac{1+\epsilon}{R} m(V)$, for each j. The *balanced partitioning* problem asks us to produce the partition that minimizes the cost of the edges cut, subject to this balance constraint, where the cut cost is defined to be $c(\delta(\mathcal{P})) := \sum_{e \in \delta(\mathcal{P})} c_e$, and $\delta(\mathcal{P})$ contains the edges whose endpoints are in distinct clusters of $\mathcal{P}$.

Now we relate the balanced partitioning problem to cache miss rates under TARS. We explain three choices made in graph construction: the query ID node masses, edge costs, and term node masses, in that order.

Consider any partition $\mathcal{P} = \{V_0, \ldots, V_{R-1}\}$ of $T \cup I$ into R clusters, and project it onto T to produce a clustering $\{T_0, \ldots, T_{R-1}\}$ of just the term nodes, where $T_j = V_j \cap T$. Recall that the terms in cluster T_r will be affinitized to replica r by the voting table derived from these term clusters using the method described in Section 3.1. Because all query ID nodes in the graph have node weight 0, they can move freely from cluster to cluster without affecting the balance constraint. This motivates the following definition.

Definition 2. *A partition $\mathcal{P}$ is* query-optimal *if each query id $i \in I$ belongs to the cluster V_r that minimizes $\sum_{t \in q_i \setminus T_r} \mathrm{sz}_t(P_t - N_t)$.*

In other words, having fixed the clustering of the terms, each query goes with the cluster that minimizes the cut. Thus, it is trivial to convert any partition into a query-optimal one without affecting the node mass balance constraint.

Theorem 1. *Suppose partition $\mathcal{P}$ is query-optimal, and query node i is in cluster V_r. When using the voting table induced by $\mathcal{P}$ via (6), TARS routes query q_i to replica r.*

PROOF. We have

$$v^r(q_i) = \sum_{t \in q_i} w_t^r$$
$$= \sum_{t \in q_i \cap T_r} \mathrm{sz}_t(1 - P_t) + \sum_{t \in q_i \setminus T_r} \mathrm{sz}_t(1 - N_t)$$
$$= \sum_{t \in q_i} \mathrm{sz}_t(1 - P_t) + \sum_{t \in q_i \setminus T_r} \mathrm{sz}_t(P_t - N_t). \quad (7)$$

Since $\mathcal{P}$ is query-optimal, r is the replica that minimizes this last term, so it minimizes $v^r(q_i)$, so TARS selects r. □

This motivates setting the query node masses to zero.

Now, suppose that the queries received at the root are drawn i.i.d. from the queries in our log (with multiplicity), so every query that the root might see is represented by a node on the right side of our graph. Under this query arrival model, the expected cache page misses per query is proportional to a constant plus $c(\delta(\mathcal{P}))$, under the assumed hit rates P_t, N_t:

$$E_I[\text{cache page misses}] = \frac{1}{|I|} \sum_{\substack{i \in I, t \in q_i, \\ r = \arg\min_r v^r(q_i)}} \mathrm{sz}_t(1 - p_t^r) \quad (8)$$
$$= \frac{1}{|I|} \sum_{i \in I, t \in q_i} \mathrm{sz}_t(1 - P_t) + \frac{1}{|I|} \sum_{\substack{i \in I, t \in q_i: \\ C(t) \neq C(i)}} \mathrm{sz}_t(P_t - N_t).$$

The first term is a constant that doesn't depend on the clustering, and the second term is $\frac{1}{|I|}$ times the cut size.

Finally, we must explain the term node masses. Recall that the R leaf replicas serving a given shard are all identical, and in particular their RAM cache has the same size. Although the cache size does not appear explicitly in the graph model, it is present implicitly. In particular, since the query stream is assumed to be i.i.d., the cache *hit* rate for term t on replica r is the same as its cache *occupancy* rate. Therefore, the total cache space occupied on replica r is:

$$\sum_{t \in T} \mathrm{sz}_t p_t^r = \sum_{t \notin T_r} \mathrm{sz}_t N_t + \sum_{t \in T_r} \mathrm{sz}_t P_t$$
$$= \sum_{t \in T} \mathrm{sz}_t N_t + \sum_{t \in T_r} \mathrm{sz}_t(P_t - N_t). \quad (9)$$

The first term in (9) is a constant, while the second is $m(T_r)$. At steady state, every cache is always completely full. This means that $m(T_r)$ should be the same for every replica.

This entire discussion relied on the assumption that P_t and N_t are the cache hit probabilities for term t on its preferred and non-preferred replicas, according to the clustering. This assumption puts the cart before the horse, in the sense that the clustering determines the voting table, which determines the p_t^r. So there is no theorem here, only intuition for why a balanced partition that minimizes the cut constitutes a good clustering from which to build a voting table. In Section 5, we demonstrate that a small-cut balanced partition of the graph corresponding to $P_t = 1, N_t = 0$ for all $t \in T$ yields a voting table that substantially decreases the cache miss rate compared to our baseline. Moreover, we show that the cut cost is positively correlated with the cache miss rate, so we would expect a clustering that does better on the cut minimization objective will also lead to a better voting table. To produce such a clustering, we apply a distributed balanced partitioning algorithm that our team developed earlier [6].

4. ITERATIVE REFINEMENT

The previous section gave a method for learning good weights for the TARS voting table. We now show how to use a cache simulator to improve those weights.

Given an initial table of weights w_t^r, we run a query log through a simulator as described in Algorithm 1 (Section 5), using TARS with voting weights w_t^r as the replica selection policy. However, we record one additional bit of data. In Algorithm 1, we record the number of cache page hits and misses every time a term t is sent to a replica. For purposes of iterative refinement, we additionally peek inside the cache on each of the other replicas to see if that term *would have* incurred a cache hit or miss. We then use all of this data to compute an empirical cache hit probability p_t^r for each term t and replica r. Since we peek into cache r no matter whether the term is actually sent to that replica or not, each p_t^r is a fraction whose denominator is the number of times term t appears in the log.

We then compute new weights $\hat{w}_t^r = \mathrm{sz}_t(1 - p_t^r)$, as in (4), and update our full set of weights to be a convex combination of the old and new weights: $w_t^r \leftarrow (1 - \tau) w_t^r + \tau \hat{w}_t^r$, where $\tau \in (0, 1]$ is a *step size* parameter. We can iterate this process as many times as we like on a set of training logs. For each of these successive voting tables, we can then evaluate the empirical cache miss rate on a separate set of validation logs to detect overfitting.

5. EXPERIMENTS

We now describe a set of experiments to show the efficacy of TARS, exploring different cache sizes, numbers of replicas, and methods of building the voting table. All experiments are performed using *real query logs* gathered from our production web search backend. We evaluate cache miss rates via simulation of the replica selection process and the caches on each replica of a single shard.

This simulation approach offers several advantages over running full-cluster loadtests. First, it allows us to experiment with different cache sizes and different numbers of replicas, instead of being restricted to the values in use by our production system. Second, it allows a laser focus on the aspects of the production system that are directly related to this change, stripping away the many confounding factors that invariably add noise to experimental results in any real system. Third, it is more computationally efficient by many orders of magnitude. Were it even possible, running all these experiments on the full cluster would be expensive. During our development of TARS, we conducted extensive experiments comparing our simulator with our production clusters, and found close agreement between the two.

We identify a single run of the simulator by a 5-tuple of parameters $(L, \mathrm{EvP}, S, R, x)$, where the symbols mean:

- L : query log to use (Section 5.1)
- EvP : cache eviction policy (always LFU)
- S : size of the cache on each replica (Section 5.2)
- R : number of replicas, $\in \{1 \ldots 5\}$.
- x : the replica selection policy.

Section 5.1 describes the query logs L, and Section 5.2 describes how we choose cache sizes S. Whenever a new item needs to be pulled into an already-full cache, the eviction

policy tells which items should be removed to create the necessary space. Although we state this as a parameter of the simulator because any eviction policy could be used, in all of the experiments that we report, we just use the LFU policy, which evicts the item that has been *least frequently used* since entering the cache. In our case, PL_t was "used" if the replica received a query containing term t. We also tried LRU (*least recently used*), and the results were nearly identical. Our production system uses a different eviction policy, but also gives similar results.

The replica selection policy can either be FPT, or TARS using some specific voting table.

Each of our simulations entails running a query log through a root simulator and R leaf cache simulators. Starting with empty caches at each replica, we run log L through the simulator once to warm up the caches, then run it through a second time to actually count the cache page hits and misses at each leaf. Algorithm 1 details this process.

Algorithm 1 Simulate(L, EvP, S, R, x)

```
Initialize empty cache^r, r = 0, ..., R − 1
for phase ∈ [warmup, measure] do
  for q ∈ L do    ▷ iterate over queries in log
    r ← x(q)    ▷ select replica via policy x
    for t ∈ q do    ▷ iterate over terms in query
      if t ∈ cache^r then    ▷ t hits cache
        Hit(cache^r, t, EvP)    ▷ update EvP's state
        if phase = measure then
          record sz_t page hits on cache^r
      else    ▷ t misses cache
          ▷ EvP evicts just enough PLs to insert PL_t
        EvictAndInsert(cache^r, t, EvP)
        if phase = measure then
          record sz_t page misses on cache^r
return page accesses & misses for each term t, replica r
▷ miss rate = misses / accesses
```

We evaluate the effectiveness of the policy x in these simulation scenarios via its aggregate cache page miss rate, which is defined to be the total page misses divided by total page accesses recorded by the simulator in its measurement phase, where both quantities are summed over all R replicas. For this purpose, we ignore the cache miss rates on the individual replicas (but see Section 9).

In the next subsections, we first describe our query logs via some revealing statistics (Section 5.1), then explore the tradeoff between cache size and miss rate in order to select an interesting set of cache sizes to use (Section 5.2). We evaluate our balanced partitioning method for constructing binary voting tables, comparing it to several alternatives (Section 5.3), then report on iterative refinement, including a crude investigation of the step size parameter, and quantifying the resulting caching improvements (Section 5.4). Finally, we examine the robustness of our cost tables by training them on queries from one continent and testing on another (Section 5.5).

One important point that runs throughout this analysis is that all of our voting table methods improve as R increases, whereas the baseline FPT method does not. This adds a pleasant feature to our search clusters that did not exist before: adding replicas to keep pace with an increased serving load automatically helps to alleviate the flash IO bottleneck.

5.1 Query logs

We collected 9 independent query logs, 3 each from Asia (AS), Europe (EU), and North America (NA), meaning that essentially all of the queries in the log originated from that geographic region. We filtered out all terms that are pinned in RAM in our production system, since these terms are irrelevant to the flash IO bottleneck. If the query becomes empty after filtering, we discard it. We denote the resulting logs by {AS,EU,NA}-{train,valid,test}, and use them as training, validation, and test sets, respectively. Table 1 summarizes these logs with some interesting statistics.

Table 1: Query log stats.

log L	Q	$\frac{DQ}{Q}$	$\frac{N_1}{N}$	$\frac{N_1}{TA}$	$\frac{TP_1}{TP}$	$\frac{TP}{S_5}$
AS-train	8.84M	91.2%	64.8%	12.4%	.10%	3914
AS-valid	8.54M	91.4%	65.0%	12.6%	.10%	3779
AS-test	8.54M	91.4%	65.0%	12.6%	.10%	3779
EU-train	8.66M	91.4%	67.0%	16.0%	.17%	2600
EU-valid	8.37M	91.5%	67.2%	16.2%	.17%	2511
EU-test	8.40M	91.5%	67.2%	16.2%	.17%	2520
NA-train	7.61M	91.4%	67.1%	16.0%	.15%	1967
NA-valid	7.38M	91.5%	67.2%	16.2%	.16%	1910
NA-test	7.42M	91.5%	67.2%	16.2%	.16%	1919

In the table, Q and DQ denote the number of queries and distinct queries (resp.), N the number of distinct terms, TA the total number of term accesses (i.e., terms with multiplicity), and TP the total number of page accesses (i.e., TA weighted by sz_t). The subscript "1" means that we restrict the quantity to the set of terms N_1 that appear exactly once in the log, while S_5 denotes the largest cache size that we use with this log (Section 5.2).

Notice that only ∼9% of queries are repeats, while roughly 2/3 of the terms in each log appear only once. These are indications of the heavy tail of the search term distribution. The N_1 terms that appear only once can never hope to hit cache for any reasonable eviction policy. The bad news is that these constitute 12-16% of the term accesses. The good news is that these rare search terms tend to also have shorter PLs, so although we expect $\frac{TP_1}{TP}$ to be a lower bound on the cache miss rate, at $< 0.2\%$, this is a non-issue. With the baseline policy, $S = S_5$ yields a cache miss rate of 5% (by construction in Section 5.2); since $\frac{TP}{S_5} \geq 1910$, our warmup phase turns over the cache at least 95 times, which is a thorough warmup by any measure. Although we sympathize with the reader's probable desire to know some other data such as N and $\frac{TA}{Q}$ (the average number of terms per query), we must withhold them for business reasons. These stats may not be reflective of user queries for multiple reasons, including filtering of pinned terms and filtering by caches higher in the search stack.

5.2 Cache sizing

There is an inverse relationship between cache size and miss rate: a larger cache results in fewer important PLs being evicted, lowering their miss rates. For cache sizes S spanning a wide range, we ran log NA-test through a single replica and plotted the cache miss rates in Figure 5. By inverting the function represented by this curve, we can compute the cache size achieving any target miss rate M.

There is a sharp "knee" in the curve right around miss rate 20%. The curves for AS and EU are similar. Thus, the

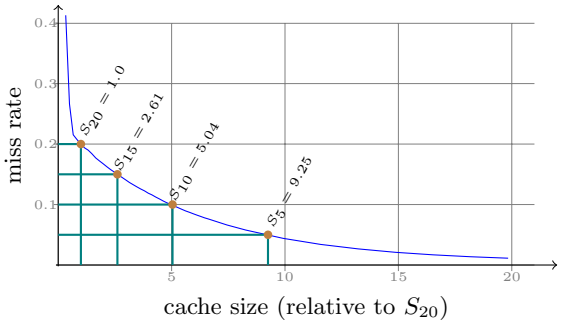

Figure 5: Tradeoff between cache size and miss rate.

benefits of increasing the cache size are enormous until the miss rate drops to around 20%, then they level off somewhat.

This suggests 20%, 15%, 10%, and 5% as sensible target miss rates. For each log in {AS, EU, NA}-test, we build an analogous curve. The cache size that causes LFU to achieve a miss rate equal to M we denote by $S_M(L)$, or just S_M if L is understood. We use binary search to compute S_{20}, S_{15}, S_{10}, and S_5. The resulting sizes appear in Table 2. In order to avoid leaking details of our production system, we always scale the cache sizes for each region so that $S_{20} = 1$.

Table 2: Larger caches show diminishing returns.

	cache size S			
region	S_{20}	S_{15}	S_{10}	S_5
AS	1	2.05	3.73	6.94
EU	1	1.73	3.50	6.83
NA	1	2.61	5.04	9.25

5.3 Binary tables

We now arrive at the punchline: the effect of TARS on cache miss rates, summarized in Table 3. Ignore the columns marked "+IR"; we will return to those in Section 5.4.

Each row corresponds to a different log and a different number of replicas. In all cases, we use our *-train log to train our tables of voting weights, then run the cache simulation on the *-test log and report those results. We used cache size S_{10} throughout.

The column marked FPT selects replicas via query fingerprint, which is our baseline method since that is what our system used prior to our introduction of TARS.

The three columns marked "binary" are all cost tables derived from the "wrong-but-useful" assumption that $p_t^r = 1$ for term t's preferred replica and 0 o.w. The RND column corresponds to a random clustering: i.e., each term chooses its preferred replica randomly. Actually, we performed ten random clusterings, evaluated them all, and report the best of the ten. However, there was not much variance among these RND tables. For instance, for NA-5, the RND voting tables gave miss rates ranging from 6.08% to 6.18%.

The column marked DC (for "diversified caching") is an adaptation of an algorithm suggested by Xu et al. [82]. They study the problem of building different static caches for each replica. They examine several variants of their algorithm, and report the best performance for one they refer to as $D^2 - DC$, so this is the version we implemented. In their setup, some terms t could have more than one preferred replica. When building our voting table, we set $p_t^r = 1$ for each such r, and 0 for the others.

The column marked BP corresponds to our clustering based on balanced partitioning of the term-query graph constructed from the training logs (Section 3). However, we prune the voting tables to contain only the terms that appear at least 4 times in the training log, to reduce noise.

How do these compare to the best cache miss rate we could possibly achieve? Let us compare two scenarios: using the best possible TARS table with R replicas, or running all queries through a single cache that is R times the size. In general, resource pooling should always be better, so the second miss rate should serve as a lower bound for the first, and we display it in the table as column Unif (for "unified" cache). For $R = 2$, BP is about halfway or more from FPT to the lower bound.

Table 3: Cache miss rates in % show impact of TARS. Uses cache size S_{10}.

		binary			+IR		
log-R	FPT	RND	DC	BP	RND	BP	Unif
AS-2	10.39	8.56	7.80	6.75	8.16	6.63	4.58
AS-3	10.36	7.48	5.78	5.94	6.86	5.70	2.27
AS-4	10.35	6.91	5.46	4.85	6.07	4.08	1.24
AS-5	10.33	6.58	5.08	4.46	5.61	3.59	0.77
EU-2	10.12	8.18	8.43	7.54	7.91	7.44	4.92
EU-3	10.11	7.30	7.28	6.27	6.84	6.15	2.66
EU-4	10.15	6.78	6.70	5.77	6.12	5.35	1.62
EU-5	10.14	6.35	5.92	5.39	5.58	4.81	1.09
NA-2	10.00	7.96	7.61	6.66	7.70	6.59	4.30
NA-3	10.00	7.03	6.45	5.43	6.46	4.90	2.03
NA-4	9.97	6.51	5.61	4.37	5.64	3.77	1.09
NA-5	10.00	6.08	4.74	3.98	5.17	3.30	0.67

Several things jump out from this table. First, even the "dumb" RND clustering is a good idea, compared to the baseline FPT replica selection method. It is worth repeating that this baseline is not just a straw man: it is the actual method that our system always used in production, until the TARS innovation. Second, the BP voting table reduces cache misses by 25% to 35%, even with only 2 replicas. With $R = 5$, the win soars to the [47%,60%] range.

We would like to know whether BP works well because balanced graph partitioning is actually a good model, or if it was just good luck. To discern, we start with the clustering from our BP method, and create a sequence of degraded clusterings. For each degraded clustering, we choose a parameter $f \in [0, 1]$. We flip a biased coin with heads probability f for each term node $t \in T$. If heads, we reassign t to a cluster chosen u.a.r. If tails, t remains with its initial cluster. When $f = 0$, nothing changes, so we got the same clustering that the BP gave. When $f = 1$, the original clustering is ignored, so we get a uniform random clustering as in the RND column of Table 3. Intermediate values of f interpolate between these extremes. We tried this for $f = 0, 0.001, 0.002, \ldots, 0.999$, built the corresponding voting tables, and evaluated the cache miss rates. Figure 3 shows a scatter plot, using NA-train to train the BP tables with $R = 5$, and NA-test to evaluate the results. Although there is some dispersion, the plot shows a nearly linear relationship between the graph cut cost and the cache miss rate, exactly as the theory predicted in equation (8).

5.4 Iterative refinement

For our iterative refinement technique of Section 4, one potentially important choice is the step size parameter τ. We performed a somewhat crude experiment to guide our choice there. Starting from the voting weights given by the BP method for the NA logs and 3 replicas, we applied 30 rounds of the iterative refinement method. We then evaluated the cache miss rate on both the training logs (Figure 6) and the validation logs (Figure 7). The plots show the evolution of the cache miss rate by iteration, for $\tau = 0.25, 0.5, 1$. The two plots are on the same scale, to aid comparison.

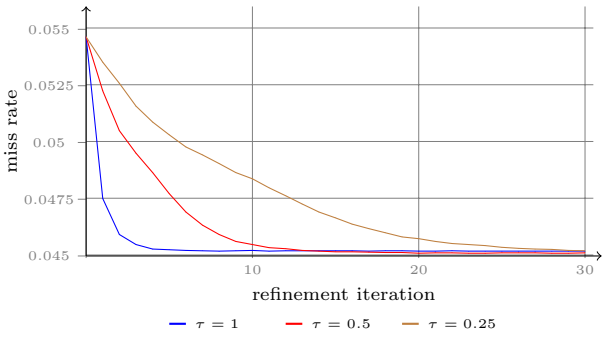

Figure 6: IR, cache miss rate on NA-train.

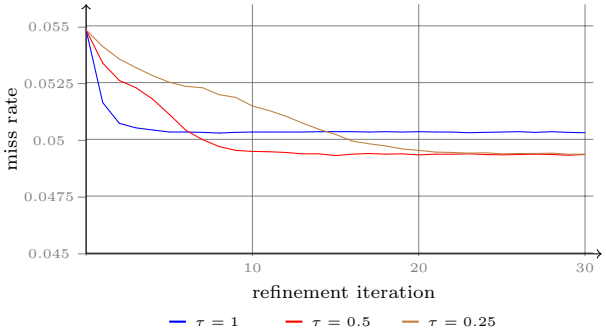

Figure 7: IR, cache miss rate on NA-valid.

Oddly, all three values of τ converge to the same miss rate on NA-train. However, on NA-valid, $\tau = 0.25$ and 0.5 converge to a better value than $\tau = 1$. For $\tau = 0.5$, the table converged by around 10 iterations. Thus, for our iterative refinement experiments (the "+IR" columns of Table 3), we perform 20 iterations with $\tau = 0.5$, just to be safe.

There are several take-home points from Table 3. First, iterative refinement improves RND more than BP. However, the starting point matters: even plain BP is better than RND + IR. Also, the improvement from IR increases with the number of replicas R.

5.5 Geographic generalization

When we first began work on TARS, we worried that the term-query graph would cluster along language barriers. What would happen if we trained the voting tables on EU logs, and the balanced partitioner assigned all of the English terms to replica 0, and all of the Chinese, Japanese, and Korean terms to replica 1? In our NA search clusters, the vast bulk of the traffic would prefer replica 0, whereas in the AS search clusters, the vast bulk would prefer replica 1. This would eliminate the kind of cache specialization that TARS is supposed to induce. Fortunately, this has not been our experience in practice. One explanation is that the Jaccard similarity of every pair of logs is greater than $\frac{1}{2}$ (Table 4).

To illustrate this effect, we fixed $R = 5$ and used the BP method to train 3 different voting tables, using {AS, EU, NA}-train. We then evaluated them on each of the three logs {AS, EU, NA}-test. Table 4 shows the results. As expected, each region achieves its lowest miss rate using the table trained on that region, with the exception of AS, which actually improves slightly when using tables trained on NA. Although we see degradations up to 1.16x, this still leaves a substantial win, relative to FPT. The "Jaccard" columns show the weighted Jaccard similarity between the vectors of terms contained in the two given logs, where the weight of each term t in log L is $\text{sz}_t \cdot$ (# occurrences of t in L).

Table 4: Cache miss rates if we serve using voting tables trained with BP on a different continent.

	train miss rate			train Jaccard		
test	AS	EU	NA	AS	EU	NA
AS	1x	1.157x	0.930x	0.957	0.528	0.610
EU	1.050x	1x	1.017x	0.511	0.956	0.734
NA	1.128x	1.161x	1x	0.588	0.722	0.961

6. LOAD-BALANCING MULTIPLIERS

So far we have mostly ignored the load-balancing multipliers μ^r because we feel that the voting weights are the more interesting part of TARS. For completeness, we now explain the multipliers. For now, consider just a single root and the R replicas of a single shard.

The root maintains a set of non-negative weights m^r, one for each replica r, and the multiplier μ^r that appears in equation (2) is defined to be $1/m^r$. Every time one of the replicas r sends a response to the root, it piggybacks a number u^r representing its CPU utilization (or, if flash is local, the max of CPU and flash IO utilization). The root then updates m^r via the following formula:

$$m^r \leftarrow m^r + \beta(\bar{u} - u^r) \tag{10}$$

where $\bar{u} := \frac{1}{R}\sum_{r=0}^{R-1} u^r$ is the average utilization over the R replicas, and $\beta > 0$ is a tunable parameter. In other words, the root increases the weight of replicas with below-average load, and decreases it for replicas with above-average load. The multipliers m^r are normalized to sum to 1, and the update equation (10) preserves this. For a query q where $v^r(q) = 0$ for each replica r (say, because no voting weights exist for these terms), the load balancer fingerprints q, splits up the fingerprint space into R contiguous segments proportional to the m^r, and routes q to the replica in whose segment its fingerprint falls. Since the fingerprinting function is highly uniform, the m^r can therefore be thought of as probabilities, in this cache-oblivious $v^r(q) = 0$ case.

In reality, there are multiple roots and multiple shards, and each of the roots maintains its own multiplier for each replica of each shard. These multipliers are able to stay relatively in sync across the multiple roots because the serving QPS is high enough that each root talks to each leaf relatively often, so its load estimate stays fresh.

7. PRODUCTION EXPERIENCE

In the actual Google web search engine, we launched TARS in four stages. First, the cache-oblivious *dynamic load balancer* (DLB) component, i.e., TARS with an empty voting table and hence no term affinity. This dramatically reduced the dispersion of the CPU load distribution, bringing down tail latencies and thereby allowing the system to handle 4% more QPS within the acceptible latency bound. As expected, it had almost no impact on flash IO. Second, we added binary voting tables (i.e., BP), which reduced aggregate leaf flash IO by 26.5%, and leaf CPU usage by 1.5% (explained by less CPU stalling while waiting for IO). Third, we improved our probability estimates (not described in this paper), reducing flash IO by an additional 2.3%. Finally, we added iterative refinement (i.e., BP + IR), reducing flash IO by an additional 6.0%.

Recall that the two main bottlenecks in our system were CPU and flash IO. Overall, we reduced flash IO hy roughly a factor of 0.675x, corresponding to a 1.48x increase in throughput capacity on the flash IO bottleneck. Paired with unrelated changes that improved our CPU throughput, most of this 1.48x increase translated directly to higher throughput capacity for the overall web search backend.

Several comments are in order here. First, the DLB lowers latency at constant QPS, and adding term affinity further lowers the mean and tail latencies for flash reads. Thus, we do not sacrifice latency to achieve this higher throughput.

Second, the load balancing weights and the voting weights impose only trivial overheads on the roots. In terms of RAM, each root has to maintain a 4B float m_s^r for each leaf (i.e., each replica r of each shard s). It also maintains a single table of voting weights that it uses for all shards. Sufficiently rare terms are excluded from the voting table, as they would just add noise. That is why in our experiments of Section 5, we limited the voting table to terms that appeared at least 4 times in the training log, of which there were roughly 4.5M. If we assume the average term string is at most 10B long, then the voting table would take about 45MB to store the term strings (depending on the representation), plus about 18MB per replica to store the voting weights as 4B floats. This is a trivial portion of the overall RAM available at our roots. In case RAM is tight, there are compression schemes for the voting table that save a large fraction of this RAM while imposing a small CPU overhead to decompress it on lookup. Similarly, the CPU costs of updating the multipliers m_s^r via equation (10) and of looking up the voting weights w_t^r are a miniscule fraction of the work done by the root for each query.

Third, our flash optimizations target a particular cache that is the biggest consumer of flash IO. On this cache, we actually reduced cache misses by roughly a 0.5x factor, but since there are other operations unaffected by our optimization that also consume flash IO bandwidth, the overall reduction in flash IO was only the 0.675x stated above.

We performed some recent experiments on a live production cluster to reproduce these results (Figures 8 and 9). Since our system evolves rapidly, the results do not perfectly match the historical numbers cited above, but they are close. Figure 8 shows the distribution of the leaf CPU utilizations in three different scenarios: (i) even splitting by fingerprint (i.e., FPT from Section 5) (ii) cache-oblivious DLB (i.e., TARS with an empty voting table) (iii) TARS using voting tables built via BP + IR.

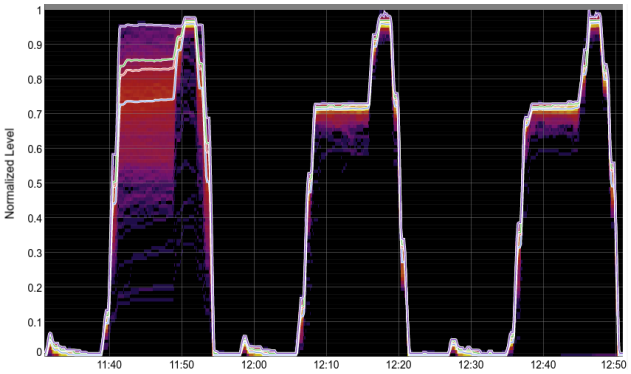

Figure 8: CPU: FPT vs. DLB vs. TARS

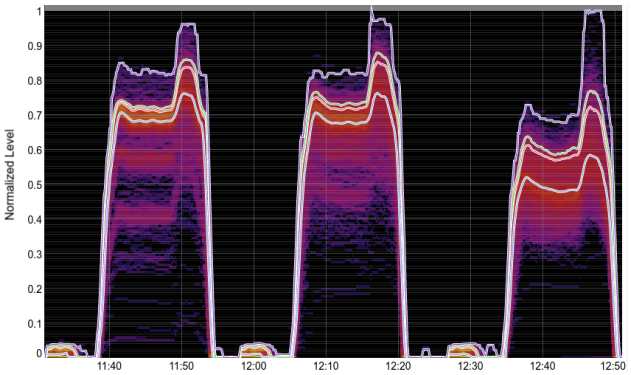

Figure 9: Flash IO: FPT vs. DLB vs. TARS

The three humps in Figure 8 show these scenarios, left to right. The x-axis shows time of day, and the y-axis shows leaf CPU utilization on an arbitrary scale. Lines showing the 50th, 90th, 95th and 99.9th percentile utilizations supplement the heatmap. Each loadtest has two phases: normal and overload (the bump at the end, which should be ignored). Figure 8 shows what we claimed above, that the DLB excels at concentrating the utilization distribution, and the addition of voting tables does not make it any worse.

Figure 9 shows the results of the same three loadtests, but plots flash IO pages/sec instead of CPU. In each of the scenarios, we see a dip at the start of the loadtest, as the cache warms up. As expected, the cache-oblivious load balancing does nothing to decrease the flash IO, but TARS decreases it dramatically, 26.8% in aggregate. Moreover, even the 99.9th percentile leaf flash IO for TARS is about the same as the median leaf flash IO without term affinity, and both the mean and tail flash read latencies are significantly reduced. In our current production configuration, our load balancing weights μ^r are based only on CPU utilization. If the flash were both local and acting as the bottleneck at each leaf, then we would expect the load balancer to make the flash IO distribution much more concentrated. Even so, the flash IO distribution is slightly tighter with TARS than without.

8. RELATED WORK

Xu et al. [82] study a similar system to ours, namely a replicated search engine architecture. They focus on designing a different static cache for each replica, and do not

mention a dynamic cache. Thus, they enforce cache specialization by fiat, whereas our approach is to induce cache specialization in a dynamic cache via the replica selection rule. Their algorithms are also completely different, and in particular do not employ graph partitioning.

A pair of papers from Facebook describe a large-scale, distributed hypergraph partitioner (Kabiljo et al. [30]) and how to apply it to route HTTP requests from Facebook users to compute clusters, so as to improve caching of objects retrieved by those requests (Shalita et al. [68]). Their users are analogous to our queries, and their objects are analogous to our terms or PLs. One big difference is that they have a known universe of users, whereas the universe of potential search queries that we must handle is essentially unbounded. Thus, they can get away with an explicit clustering of users, but we must resort to clustering our queries indirectly via our voting scheme. Their approach to balancing load is also quite different: they use 21,000 clusters of users (many more than their number of compute clusters), and shift entire user clusters when necessary to balance load. Unlike us, their graph does not explicitly model cache misses; instead they rely on the structure of the social graph being correlated with the objects they need to cache.

Although our system is doc-sharded, another option for distributing the web index is *term-sharding*, wherein each term stores its entire PL in one place, and the terms are partitioned among the machines. In the basic version, the root collects PLs from each of the leaves that hold a term from the query, then does the intersection. This puts great computational load on the root. Moffat et al. [48] suggest pipelining, where the query is bounced from leaf to leaf, each performing an intersection with the PLs it owns. In a follow-up, Moffat et al. [47] use bin packing to balance workloads across the leaves. Zhang and Suel [83] take this farther, by co-locating terms that tend to be co-queried. They model the benefits of co-locating terms, then map this heuristically and inexactly to a balanced graph partitioning problem on the terms only, which they solve with METIS [34]. Cambazoglu et al. [13] refine this by modeling the problem as balanced hypergraph partitioning, where the nodes are terms and the hyperedges are queries from a log, although they admit that even with their optimizations, term-sharding tends to be inferior to doc-sharding. There is a superficial resemblance to our bipartite term-query graph (Section 3.2), since the hyperedges can be transformed into query nodes. However, cut costs in the hypergraph and corresponding bipartite graph are not equivalent. In particular, it is essential to our modeling that the cut costs induce each query node to select the same replica in the partition that it would under TARS.

Caching is vital throughout the web search stack. In some layers, it can be useful for caching policies to take costs into account, not just access frequencies. Ozcan et al. [54] consider caching 5 types of objects: documents, PLs, intersections of PLs, relevance scores, and full query results. They co-locate the root and leaf jobs so that these 5 object types can share a single cache. They compute a gain := cost saving × frequency / size for each object, and fill this static cache greedily (offline). Ozcan et al. [55] addresses just query result caching. They examine static, dynamic, and hybrid static-dynamic caching, considering several simple cache filling / eviction policies that take into account the CPU and spinning disk cost of executing a query. Neither of these works is directly relevant to ours, since they rely on the gain parameters varying widely, whereas the cost per page of retrieving a PL from flash is constant, so for us, gain reduces to frequency. TARS could easily incorporate query CPU costs, but these are difficult to predict. Moreover, Figure 8 suggests there is limited room for improvement.

In the early Internet, DNS servers were perhaps the only big distributed serving systems. Traffic growth soon motivated replication of web servers, FTP mirrors, and CDNs [18, 51]. Other *server farms* such as build farms [59, 66, 60, 38], and render farms [19] as well as many cloud services [2, 25, 28, 44, 53] followed suit. Load balancing is a vital component of these systems. Balancing can be static or adapt to changes in traffic and/or replica failures [80]. It may be done in hardware [43, 58, 22] or via software [63, 71].

Four types of web server replica selection were studied in [14]: the decision is made by the client, the DNS server, a central dispatcher, or by the replicas themselves [75, 70]. Several routing strategies have been studied when a central authority is in charge of load balancing, which is typically the case for web servers and distributed databases. Well-known methods in database systems include *list partitioning* (e.g., based on geography [15]), *range partitioning* (e.g., based on time or zipcode), *round-robin partitioning*, *hashing-based partitioning* and composite methods [56].

Certain applications (e.g., serving session-aware web content) benefit from consistently routing specific requests (of the same HTTP session) to the same replica. This requirement, known as *persistence* or *stickyness*, is supported by most commercial and free web servers [71, 10] or proxy servers [63] today. Our TARS method is similar to persistent load balancing, in that we aim to consistently route repeated occurrences of the same term to the same replica. However, the two applications differ because we route multi-term *queries* as an atomic whole, but it is the *terms inside queries* that are the keys whose values are to be cached.

Persistent load balancing was extended [45] to have guarantees on the maximum load. The proposed algorithm was implemented in HAProxy [76] and deployed to Vimeo servers, which reduced their maximum memcached server bandwidth at peak hours by a factor of ∼8x [65].

Distributed hash tables (DHTs) store large numbers of key-value pairs and many implementations use persistent load balancing to optimize for fast access time. This has found application in many peer-to-peer network architectures [31, 67, 74, 84] as well as in fast implementation of distributed graph algorithms [6, 8, 35, 73, 64]. Stoica et al. [74] propose a simple, scalable method (called "Chord") for mapping keys to peers using $O(\log n)$ virtual nodes per real node to achieve the desired load balance. Byers et al. [12] apply the "power of two choices" paradigm [46] instead of the virtual nodes to improve Chord's performance [74]. Roughly speaking, their solution shaves a factor of $\log n / \log \log n$ from the memory usage of the DHT. Karger and Ruhl [33] propose two new load-balancing protocols: One fixes the $O(\log n)$ space and bandwidth blow-up of Chord [74] while maintaining its look-up and load-balance guarantees (improving over [12]); the other protocol allows for range searching. Bienkowski et al. [9] show how to maintain load balance when peers join and leave arbitrarily. Their approach has less "migration cost" than that of [33]. Goel et al. [24] empirically study several consistent hashing strategies in DHTs.

Web caching is a prime application of DHTs [32, 27]. Software packages such as memcached, ElastiCache, Redis and

OpenStack's Swift are deployed in many web applications and cloud platforms [4, 52, 81, 21, 42]. Shen et al. [69] study how to improve flash-based key-value cache systems such as Facebook's McDipper [41] and Twitter's Fatcache [20] by considering some of the details of flash drives, rather than treating them just as fast storage. Distributed databases also benefit from consistent hashing [37, 50, 16]. There has been extensive study for hardware and software load-balancing architectures and algorithms to manage cloud platforms [1, 22, 58, 79, 26, 62, 43]. Some works, in particular, focus on applications of consistent hashing [80, 49, 32].

Our TARS method relies on algorithms for the balanced partitioning problem on graphs (Section 3.2) to partition search terms into R clusters. Even for small instances and $R = 2$, the problem is challenging to solve [3], as it captures the *graph bisection* problem [23], known to be NP-hard to approximate within a constant factor [17]. Sub-logarithmic approximation algorithms exist [5], but are based on solving semidefinite programs (SDPs), which is impractical for datasets of decent size. Under some assumptions, Stanton [72] showed that achieving formal approximation guarantees is information theoretically impossible.

Given the hardness of this problem, there is a large body of work on heuristics for (distributed) balanced graph partitioning, including FENNEL [77], Spinner [40], METIS [34], ja-be-ja [61], ParaMETIS [57], a method based on embedding the graph on a line [6], and one based on label propagation [78]. Various graph processing frameworks have been proposed for distributed problems [39, 36]. Even simple graph problems with linear-time (or sublinear) algorithms are sometimes hard to solve for such large graphs. For example, developing distributed algorithms for the seemingly simple problem of computing connected components has attracted a large body of research recently [64, 35, 73]. The great progress in this area enabled our balanced partitioning algorithm [6], which in turn greatly improved TARS.

9. FUTURE WORK

TARS is not particular to web search, so we hope to see diverse applications. Any distributed system with the following attributes could benefit from our approach: (i) the request load is high enough that it must be distributed among multiple serving replicas, (ii) each request contains one or more lookup keys, and the replica must retrieve data belonging to each of these keys from a key-value store, and (iii) each replica uses caching to avoid some lookups.

One big open question is to find an even better way to train the voting tables. Although our methods achieved a huge improvement for our web search backend, they are still some distance from the lower bounds in Table 3, especially for large R. In particular, we are keen to try reinforcement learning approaches, but would first need to extend them to deal with the feedback loop described in Section 2. In this process, it would help if we could learn a concise and fast-to-compute model of the cache, to avoid having to run our computationally-intensive simulations.

In this paper, we focused only on the aggregate page miss rate over all replicas. If the flash that stores the PLs is remote from the serving replicas, as depicted in Figure 1, this is clearly the quantity of interest. If the flash is located locally on each replica, things become more complicated. In that case, we care not only about minimizing the *aggregate* cache miss rate, but also the *maximum* cache miss rate among the R replicas. Even in this case, we argue that our approach of minimizing the sum instead of the max is a reasonable approach. This is because multiplication by the replica weights μ^r provided by the feedback-based part of the TARS approach (Section 2) guides us to shed load on the queries that are predicted to be roughly as cheap on another replica. So the full TARS mechanism can even out imbalances that might exist if all multipliers μ^r were fixed at 1. Figure 9 shows that TARS already improves the upper tail of the flash IO distribution, even in an environment where the multipliers μ^r are being governed exclusively by CPU. Presumably, it would do even better if the system were flash-bound. That said, in a flash-bound system it might be useful to train our voting tables in a fashion that tries explicitly to balance flash IO among replicas, since that would allow the μ^r multipliers to stay closer to 1, thereby shifting less traffic away from its terms' preferred replicas.

Recall that TARS greedily chooses the replica minimizing the expected number of immediate cache page misses, given p_t^r. Stepping back, one might reasonably ask why we need to estimate these probabilities p_t^r at all. Indeed, the presence or absence of each term in the cache is known by the leaves, so an alternative approach is for the roots to simply probe all replicas to determine whether each term in the current query is present in cache.

There are several drawbacks to this strategy, which made it a non-starter for our system. First, this adds a substantial communication overhead between roots and leaves. Instead of sending just one RPC (per shard) in each direction, the root must send an RPC to each of the R replicas and wait for responses, which adds latency to the critical path. More interesting is the fact that the caches need to be warmed up before we would expect this replica selection method to work well. How should we perform this warmup? Our best method for doing that is to use our voting tables.

Lastly, it might be fruitful to consider not just static voting tables as we did in this work, but also dynamic voting tables or other replica selection policies that update over time, attempting to learn a model of the current contents of each replica's cache. This might allow us to implement an approximation to the greedy replica selection that we just discussed and shot down, by ridding ourselves of the need to probe each replica on each query.

10. ACKNOWLEDGMENTS

We thank Bartek Wydrowski and Morteza Zadimoghaddam, whose work with some of us on the DLB paved the way for TARS, and Cristi Estan, Dan Meredith, Vishal Misra, Dina Papagiannaki, and Flip Korn for helpful feedback.

11. ADDITIONAL AUTHORS

Richard Zhuang (Google, `xzhuang@google.com`).

12. REFERENCES

[1] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. Conga: Distributed congestion-aware load balancing for datacenters. *SIGCOMM Computer Communication Review*, 44(4):503–514, Aug. 2014.

[2] Amazon Web Services (AWS). `https://aws.amazon.com`.

[3] K. Andreev and H. Räcke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[4] J. Arnold. *OpenStack Swift: Using, Administering, and Developing for Swift Object Storage*. O'Reilly Media, Inc., 1st edition, 2014.

[5] S. Arora, S. Rao, and U. V. Vazirani. Expander flows, geometric embeddings and graph partitioning. *Journal of the ACM*, 56(2), 2009.

[6] K. Aydin, M. Bateni, and V. S. Mirrokni. Distributed balanced partitioning via linear embedding. In *Proceedings of the 9th ACM International Conference on Web Search and Data Mining, WSDM '16, San Francisco, CA, USA, February 22-25, 2016*, pages 387–396, 2016.

[7] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[8] M. Bateni, S. Behnezhad, M. Derakhshan, M. Hajiaghayi, R. Kiveris, S. Lattanzi, and V. S. Mirrokni. Affinity clustering: Hierarchical clustering at scale. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems, NeurIPS '17, 4-9 December 2017, Long Beach, CA, USA*, pages 6867–6877, 2017.

[9] M. Bienkowski, M. Korzeniowski, and F. M. a. der Heide. Dynamic load balancing in distributed hash tables. In M. Castro and R. van Renesse, editors, *Peer-to-Peer Systems IV*, pages 217–225, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[10] R. B. Bloom. *Apache Server 2.0: The Complete Reference*. Osborne/McGraw-Hill, Berkeley, CA, USA, 2002.

[11] G. E. P. Box. Robustness in the strategy of scientific model building. In G. N. W. R.L. Launer, editor, *Robustness in Statistics*, pages 201–236. Academic Press, 1979.

[12] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In M. F. Kaashoek and I. Stoica, editors, *Peer-to-Peer Systems II*, pages 80–87, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[13] B. B. Cambazoglu, E. Kayaaslan, S. Jonassen, and C. Aykanat. A term-based inverted index partitioning model for efficient distributed query processing. *ACM Transactions on the Web*, 7(3):15:1–15:23, 2013.

[14] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, May 1999.

[15] V. Cardellini, M. Colajanni, and P. S. Yu. Geographic load balancing for scalable distributed web systems. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS '00*, pages 20–, Washington, DC, USA, 2000. IEEE Computer Society.

[16] J. L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[17] S. Chawla, R. Krauthgamer, R. Kumar, Y. Rabani, and D. Sivakumar. On the hardness of approximating multicut and sparsest-cut. *Computational Complexity*, 15(2):94–114, 2006.

[18] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, Sept. 2002.

[19] A. Faier. Digital slaves of the render farms?: Virtual actors and intellectual property rights. *Journal of Law, Technology & Policy*, 2004(2):321–343, 2004.

[20] `https://github.com/twitter/fatcache`.

[21] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5–, Aug. 2004.

[22] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. *SIGCOMM Computer Communication Review*, 44(4):27–38, Aug. 2014.

[23] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[24] P. Goel, K. Rishabh, and V. Varma. An alternate load distribution scheme in DHTs. In *IEEE International Conference on Cloud Computing Technology and Science, CloudCom '17, Hong Kong, December 11-14, 2017*, pages 218–222, 2017.

[25] Google Cloud Platform. `https://cloud.google.com`.

[26] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based load balancing for fast datacenter networks. *SIGCOMM Computer Communication Review*, 45(4):465–478, Aug. 2015.

[27] S. Huq, M. Z. Shafiq, S. Ghosh, A. R. Khakpour, and H. Bedi. Distributed load balancing in key-value networked caches. In *37th IEEE International Conference on Distributed Computing Systems, ICDCS '17, Atlanta, GA, USA, June 5-8, 2017*, pages 583–593, 2017.

[28] IBM Bluemix. `https://bluemix.net`.

[29] Internet Live Stats. Google search statistics. `http://www.internetlivestats.com/google-search-statistics`.

[30] I. Kabiljo, B. Karrer, M. Pundir, S. Pupyrev, A. Shalita, Y. Akhremtsev, and A. Presta. Social hash partitioner: A scalable distributed hypergraph partitioner. *PVLDB*, 10(11):1418–1429, 2017.

[31] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing, STOC 1997*, pages 654–663, New York, NY, USA, 1997. ACM.

[32] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11-16):1203–1213, May 1999.

[33] D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 36–43, New York, NY, USA, 2004. ACM.

[34] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[35] R. Kiveris, S. Lattanzi, V. S. Mirrokni, V. Rastogi, and S. Vassilvitskii. Connected components in mapreduce and beyond. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC '14, Seattle, WA, USA, November 03 - 05, 2014*, pages 18:1–18:13, 2014.

[36] J. Koch, C. Staudt, M. Vogel, and H. Meyerhenke. An empirical comparison of big graph frameworks in the context of network analysis. *Social Network Analysis and Mining*, 6:1–20, 2016.

[37] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40, Apr. 2010.

[38] The Launchpad build farm. `https://launchpad.net/builders`.

[39] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD '10, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146, 2010.

[40] C. Martella, D. Logothetis, A. Loukas, and G. Siganos. Spinner: Scalable graph partitioning in the cloud. In *33rd IEEE International Conference on Data Engineering, ICDE '17*, pages 1083–1094, 2017.

[41] `https://www.facebook.com/notes/facebook-engineering/mcdipper-a-keyvalue-cache-for-flash-storage/10151347090423920`.

[42] memcached: a distributed memory object caching system. `https://memcached.org`.

[43] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 15–28, New York, NY, USA, 2017. ACM.

[44] Microsoft Azure. `https://azure.microsoft.com`.

[45] V. S. Mirrokni, M. Thorup, and M. Zadimoghaddam. Consistent hashing with bounded loads. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '18, New Orleans, LA, USA, January 7-10, 2018*, pages 587–604, 2018.

[46] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, Oct. 2001.

[47] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '06, Seattle, Washington, USA, August 6-11, 2006*, pages 348–355, 2006.

[48] A. Moffat, W. Webber, J. Zobel, and R. A. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, 10(3):205–231, 2007.

[49] M. Nasri and M. Sharifi. Load balancing using consistent hashing: A real challenge for large scale distributed web crawlers. In *23rd International Conference on Advanced Information Networking and Applications, AINA '09, Workshops Proceedings, Bradford, United Kingdom, May 26-29, 2009*, pages 715–720, 2009.

[50] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[51] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network: A platform for high-performance internet applications. *SIGOPS Operating Systems Review*, 44(3):2–19, Aug. 2010.

[52] Openstack swift: Powering the world's largest storage clouds. `https://www.swiftstack.com/product/openstack-swift`.

[53] Oracle Cloud Platform. `https://cloud.oracle.com/paas`.

[54] R. Ozcan, I. S. Altingövde, B. B. Cambazoglu, F. P. Junqueira, and Ö. Ulusoy. A five-level static cache architecture for web search engines. *Information Processing and Management*, 48(5):828–840, 2012.

[55] R. Ozcan, I. S. Altingövde, and Ö. Ulusoy. Cost-aware strategies for query result caching in web search engines. *ACM Transactions on the Web*, 5(2):9:1–9:25, 2011.

[56] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2011.

[57] D. Padua, editor. *ParaMETIS*, pages 1458–1458. Springer US, Boston, MA, 2011.

[58] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. *SIGCOMM Computer Communication Review*, 43(4):207–218, Aug. 2013.

[59] M. Pool. distcc – a free distributed C/C++ compiler system. `https://distcc.org/`.

[60] PostgreSQL BuildFarm. `https://buildfarm.postgresql.org`.

[61] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO '13, Philadelphia, PA, USA, September 9-13, 2013*, pages 51–60, 2013.

[62] C. Raiciu, F. Huici, M. Handley, and D. S. Rosenblum. Roar: Increasing the flexibility and performance of distributed search. *SIGCOMM Computer Communication Review*, 39(4):291–302, Aug. 2009.

[63] N. Ramirez. *Load Balancing with HAProxy: Open-Source Technology for Better Scalability, Redundancy and Availability in Your IT Infrastructure*. Independently Published, 2016.

[64] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma. Finding connected components in map-reduce

in logarithmic rounds. In *29th IEEE International Conference on Data Engineering, ICDE '13, Brisbane, Australia, April 8-12, 2013*, pages 50–61, 2013.

[65] A. Rodland. Improving load balancing with a new consistent-hashing algorithm. `https://medium.com/vimeo-engineering-blog/improving-load-balancing-with-a-new-consistent-hashing-algorithm\-9f1bd75709ed`, December 2016.

[66] ROS Buildfarm. `http://wiki.ros.org/buildfarm`.

[67] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001, Proceedings*, pages 329–350, 2001.

[68] A. Shalita, B. Karrer, I. Kabiljo, A. Sharma, A. Presta, A. Adcock, H. Kllapi, and M. Stumm. Social hash: An assignment framework for optimizing distributed systems operations on social networks. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI '16, Santa Clara, CA, USA, March 16-18, 2016*, pages 455–468, 2016.

[69] Z. Shen, F. Chen, Y. Jia, and Z. Shao. Optimizing flash-based key-value cache systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage '16)*, Denver, CO, 2016. USENIX Association.

[70] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, Dec. 1992.

[71] R. Soni. *Nginx: From Beginner to Pro*. Apress, Berkely, CA, USA, 1st edition, 2016.

[72] I. Stanton. Streaming balanced graph partitioning algorithms for random graphs. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '14, Portland, Oregon, USA, January 5-7, 2014*, pages 1287–1301, 2014.

[73] S. Stergiou, D. Rughwani, and K. Tsioutsiouliklis. Shortcutting label propagation for distributed connected components. In *Proceedings of the 11th ACM International Conference on Web Search and Data Mining, WSDM '18, Marina Del Rey, CA, USA, February 5-9, 2018*, pages 540–546, 2018.

[74] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Computer Communication Review*, 31(4):149–160, Aug. 2001.

[75] A. N. Tantawi and D. Towsley. Optimal static load balancing in distributed computer systems. *Journal of the ACM*, 32(2):445–465, Apr. 1985.

[76] W. Tarreau. HAProxy: The reliable, high performance TCP/HTTP load balancer. `https://www.haproxy.org/`.

[77] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. FENNEL: streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining, WSDM '14*, pages 333–342, 2014.

[78] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Web Search and Data Mining, WSDM '13*, pages 507–516, 2013.

[79] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford. Central control over distributed routing. *SIGCOMM Computer Communication Review*, 45(4):43–56, Aug. 2015.

[80] X. Wang and D. Loguinov. Load-balancing performance of consistent hashing: Asymptotic analysis of random node join. *IEEE/ACM Transactions on Networking*, 15(4):892–905, Aug. 2007.

[81] X. Wu, F. Ni, L. Zhang, Y. Wang, Y. Ren, M. Hack, Z. Shao, and S. Jiang. NVMcached: An NVM-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, pages 18:1–18:7, New York, NY, USA, 2016. ACM.

[82] C. Xu, B. Tang, and M. L. Yiu. Diversified caching for replicated web search engines. In *31st IEEE International Conference on Data Engineering, ICDE '15, Seoul, South Korea, April 13-17, 2015*, pages 207–218, 2015.

[83] J. Zhang and T. Suel. Optimized inverted list assignment in distributed search engine architectures. In *21th International Parallel and Distributed Processing Symposium, IPDPS '07, Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–10, 2007.

[84] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2001.