ROAR: Increasing the Flexibility and Performance of Distributed Search

Costin Raiciu
University College London
c.raiciu@cs.ucl.ac.uk

Felipe Huici
NEC Europe, Heidelberg
Felipe.Huici@nw.neclab.eu

Mark Handley
University College London
m.handley@cs.ucl.ac.uk

David S. Rosenblum
University College London
d.rosenblum@cs.ucl.ac.uk

ABSTRACT

To search the web quickly, search engines partition the web index over many machines, and consult every partition when answering a query. To increase throughput, replicas are added for each of these machines. The key parameter of these algorithms is the trade-off between replication and partitioning: increasing the partitioning level improves query completion time since more servers handle the query, but may incur non-negligible startup costs for each sub-query. Finding the right operating point and adapting to it can significantly improve performance and reduce costs.

We introduce Rendezvous On a Ring (ROAR), a novel distributed algorithm that enables on-the-fly re-configuration of the partitioning level. ROAR can add and remove servers without stopping the system, cope with server failures, and provide good load-balancing even with a heterogeneous server pool. We demonstrate these claims using a privacy-preserving search application built upon ROAR.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Nets**]: Distributed Systems

General Terms

Algorithms, Design

1. INTRODUCTION

Search, possibly the web's most important application, is implemented as a distributed computation over a large inverted Web index. In order to improve the performance of queries, this index is partitioned into many parts, and each part is replicated on a cluster of commodity PCs. When a query is executed, it is sent to one machine in each cluster so that the whole index is covered, and the results aggregated [5].

From a distributed algorithms point of view, which cluster each data item is stored on and which machines each query is sent to are independent of the actual *content* of the data and queries. Indeed, the algorithm is blind to this content: it is sufficient to ensure that each query reaches machines that between them hold all the data. We call this class of algorithms *distributed rendezvous*.

SIGCOMM'09, August 17–21, 2009, Barcelona, Spain.

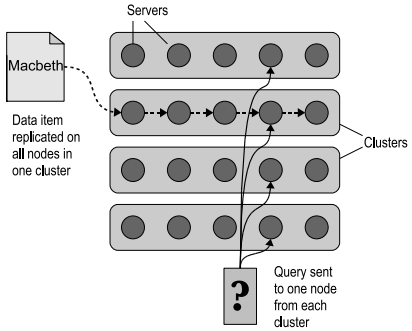


Figure 1: Basic Distributed Rendezvous

Such algorithms contrast with other more constrained look-up algorithms such as Distributed Hash Tables (DHTs), where a query is sent to precisely the node that can answer the request. To some extent, distributed rendezvous can be thought of as brute-force distributed matching. However inelegant this may seem, many real-world problems fall into this category, including web search.

Successful web search engines such as Google use parallel index-search algorithms [5], which are a form of distributed rendezvous. The datasets involved can be many terabytes in size [5], can change rapidly (consider Google News, updated continuously as news happens), and can have very high query rates. Only by spreading the search across large numbers of servers can query latency be kept low while achieving high overall throughput.

Figure 1 illustrates the basic concept. The servers are divided into clusters and each data item to be searched is replicated on all the machines in a single cluster. With this in place, a query is then sent to one machine from each cluster, thus ensuring that the query is matched against the full index. Each data entry is only matched against the query on a single machine, allowing arbitrarily complex matching rules to be performed locally. Having performed the search, each machine ranks the matches and returns the best ones. Finally, the results from all the query machines are merged, ranked once again, and returned to the user.

Given this basic strategy, the obvious question is how many nodes should be in each cluster? Each query must be sent to one node from each cluster, so increasing the number of clusters means splitting the search index into more pieces. This involves more nodes in each search, reducing the search completion time.

Although low query times are desirable, running a query on a node also has a fixed cost, as the query must be communicated to the node and a search thread instantiated there. These costs are

not negligible: if we took the extreme position of having only one node per cluster, then every node would have to try to process every query. Even though the search performed on each node would be cheap, the overall throughput would be very low.

In essence, the problem is one of balancing search latency, which benefits from a larger number of clusters, with total throughput for all nodes, which has a preference for a smaller number of clusters. A sensible strategy would be to choose the smallest number of clusters that satisfies a latency target, such as answering all queries in under a second. Once this target is satisfied, splitting into more clusters would only decrease peak throughput.

Of course, for a static data set and a constant query rate there is no great problem figuring out the number of clusters needed to satisfy a target latency, and from there to calculate the number of machines in each cluster needed to satisfy the overall throughput. However, neither the data set nor the query rate remain constant for most real applications, and the total number of machines cannot normally be changed on short timescales.

Consider again Google's search engine: over time the size of the web increases, so the size of Google's index grows. While machines can easily be added to existing clusters in order to maintain throughput, keeping search latency constant requires repartitioning the servers into more clusters.

Google does this by removing machines from an existing cluster and adding them to a new cluster configuration during a low traffic period [9]. Once this completes, the front-end load balancers start using the updated machines to answer a fraction of queries. The next batch of machines are then removed, repartitioned into the new clusters and updated, and so on. This works but is inflexible: repartitioning needs to be a rare event, and it cannot be performed in response to a load spike because it must be done at a quiet time.

In this we paper examine the question of how to change the partitioning of a *running* distributed rendezvous system. We propose a novel algorithm for distributing data and queries between servers that balances load well, and is much more amenable to on-the-fly changes to partitioning, even under conditions of heavy load. This additional flexibility can be used to cope with flash crowds, to manage data sets that change even more rapidly than Google's, and may even be used to adaptively control the total work done in such a data center so as to reduce overall demand for electrical power, an important concern for data centers these days.

2. THE NATURE OF THE PROBLEM

Let us parameterize the problem:

- Each data item is *replicated* and stored on r servers.
- Each query is run in parallel on p servers (we say the query has been *partitioned* and that p is the *partitioning level*).

The aim is to perform data replication and query partitioning such that every query meets every data item. If all data items have the same number of replicas and all queries are sent to the same number of servers, it is trivial to see from Figure 1 that with n servers it must be the case that:

$$p \cdot r = n \tag{1}$$

This characterizes the basic tradeoff in distributed rendezvous: as p increases to improve latency, r generally decreases, so a node stores less data but must handle more queries.

In reality the situation is not quite so simple, and so this provides a lower bound. If load balancing is not perfect, or if nodes fail, or just to add resilience, larger values of r may be used. Hence:

$$p \cdot r \geq n \tag{2}$$

Note that on each server, a local index (such as an inverted index) may be created based on the items (documents) assigned to that server. This index will be used to perform fast local matching. However, the latency of a match on each node will still grow with the number of documents indexed, only more slowly. Further, this does not affect the nature of the replication process *across* servers.

2.1 Constraints

If we double p, the total cost across all servers of matching a single query remains unchanged, but twice the number of servers do half the amount of work each. Normally this will reduce query delay. However, there are additional constraints that influence the tradeoff between p and r; these are the focus of this paper:

- The processing resources of each node are bounded. Doubling p also means each node must handle double the number of queries. Each additional query requires setting up a search thread, network bandwidth to communicate the query, and imposes extra context switching overhead. For an overall system running at high utilization, p cannot increase indefinitely; beyond some point nodes will saturate.
- The long-term storage (memory and/or disk space depending on the application) on each node is bounded. Thus, there is also an upper bound on r, above which the nodes cannot store their fraction of the data items.
- For a dataset that changes rapidly, increasing r means more changes must be sent to each node. This extra work reduces the capacity of each node to handle queries.

Thus, p cannot be too large lest nodes' CPUs saturate, and it cannot be too small or nodes' storage will saturate. Generally we want to choose p to be large enough to satisfy latency bounds that are determined by usability factors, but choosing a larger p than this will increase processing costs, requiring more machines to handle peak workloads. For non-peak workloads one might assume that using a larger p than necessary would not be a problem, but modern servers require significantly more energy when they are working hard (which is the case when p is increased, due to the additional per-request fixed costs incurred); for companies such as Google and Microsoft that run huge numbers of servers, minimizing power consumption is an important goal.

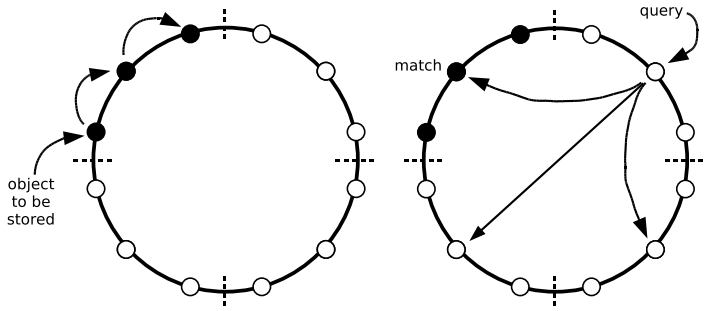
In addition to these constraints, query rates vary over time due to daily and weekly cycles as well as flash crowds. This leads to the question of whether it is feasible to change p relatively frequently. Indeed, it may also be possible to shut down or sleep nodes to save power at quiet times, and thus change r without changing p. In the next section we will examine these questions in some detail; the result will be a design for a new distributed rendezvous system that makes such changes possible at acceptable cost.

2.2 Dynamic Repartitioning

The repartitioning strategy described in Section 1, whereby during a quiet time servers are taken offline to be moved into new clusters, has several problems.

First, reducing electricity use requires running fewer servers at relatively high utilization levels rather than more servers at lower utilization.[1] Thus, reducing the capacity of the network to repartition is difficult while sustaining the required query throughput. Either the workload must have predictable quiet periods lasting for significant periods of time, or spare machines must be maintained

[1] A server requires roughly half the power when idling as when fully loaded, with the change in power between idle and loaded being fairly linear with CPU utilization



(a) Storing data items (b) Executing a query

Figure 2: A simple sliding window algorithm with p=4, r=3, and n=12.

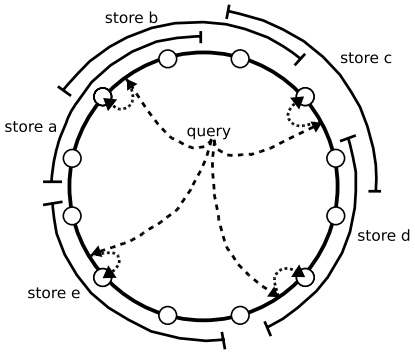


Figure 3: ROAR with n=12, p=4 and r=3. Objects are stored in arcs of length $1/p$ and queries sent to p servers at $1/p$ intervals.

and powered up during repartitioning, increasing additional infrastructure and energy costs.

Second, this strategy incurs significant data transfer costs while repartitioning. Each server will dump its current documents and reload its new part of the index. In effect, the index as a whole is copied r times, which unnecessarily wastes bandwidth and creates significant stress on the backing filesystem as servers download their new index. How big is this waste? Google reports p to be around 1,000 [3]; search is done in more than 40 data centers[18] distributed globally. In each data center the replication level is low (1-3) [10]. Let's approximate r to be 80.

The index is reportedly a few terabytes in size (for the sake of argument, let us assume it is 10TB), and thus the whole network needs to transfer around 800TB in order to repartition.

Finally, the time to repartition may be significant. An unpredicted traffic spike during repartitioning may cause an overload. Google can avoid this by repartitioning only one data center at a time and moving traffic away from that data center using DNS load balancing, but not all organizations can do so.

In both solutions above, distributed coordination is needed to decide which servers should be migrated, when, and to which cluster. Coordination is required to decide when to switch to the new configuration and when to stop the old configuration. This makes the whole process difficult to automate, cumbersome and lengthy.

All of these problems stem from the simplicity of the algorithm. Simple partitioning seems good enough in cases when r and p rarely change. On the other hand, the ability to cheaply and frequently repartition on the fly can allow a great deal of flexibility, allowing adaptation to changing operating conditions, be they due to spikes in load, changes in the data set, or equipment failure. To achieve this level of flexibility at reasonable cost we need to move away from simple partitioning strategies and examine algorithms that do not require the overlay structure to change. In the next section we introduce ROAR, a novel distributed rendezvous algorithm that meets these goals.

3. TOWARDS A SOLUTION

Our key observation is that there is no need to divide the nodes into disjoint clusters: what is important is that each data item is replicated on r nodes, and that we can arrange for every query to visit at least one of these nodes. There are random-walk algorithms that can do this [25], but they require $p \cdot r \gg n$, which is usually unacceptable. Can a deterministic algorithm do better?

The simplest solution is probably a sliding window algorithm, where the n nodes are arranged in a circle. The first data item is then stored on nodes $1...r$, the second is stored on nodes $2...(r+1)$, and the k^{th} on nodes $k...(r+k)$, with all arithmetic performed modulo n. Now if a query visits every r^{th} node it is guaranteed to reach every data item, as shown in Figure 2. Such an algorithm has some very nice properties:

- Each node stores the same number of items, and if a round-robin algorithm is used to start queries, each node handles the same number of queries (assuming r divides n precisely). In this sense it is identical to the basic partitioning scheme.
- Increasing r by one merely requires replicating each data item onto the successor node on the ring.
- Decreasing r by one merely requires deleting each data item from the node that stores it that has the greatest ID.

Thus each node plays an equal role when changing r (and consequently p). When decreasing r, no additional data needs to be copied. When increasing r by one, each node needs to copy $1/n^{\text{th}}$ of the data. During the transition, search continues to function. If r is decreasing, searches must use the new value of p during the transition to ensure correctness. If r is increasing, searches must use the old value of p until the transition is complete.

Despite these nice properties, such an algorithm has some shortcomings. First, while it works very well with a fixed number of reliable nodes, it does less well if a node fails. In this case, *all* the objects stored on the failed node need to be replicated once more, as they've just lost one replica. These replicas need to be added by the r successors of the failed node; this implies that each node needs to monitor the health of its r predecessors; for large values of r, the costs can be significant. Finally, until the new replicas are added, query execution could miss some objects.

The basic problem with this simple sliding window algorithm stems from the fact that the nodes have a discrete position on the ring. Data is then replicated across consecutive nodes holding a range of these discrete positions. If the list of nodes changes (nodes are added, shutdown to save power, or fail), this impacts the *relative* positions of nodes, and so has non-local consequences.

Beyond this, another problem is that all nodes are treated equally—also a result of the discrete nature of the node positions on the ring. In practice, it is rare that all nodes in a data center are of identical performance, as equipment tends to be purchased over time. An explicit goal is to be able to effectively utilize heterogeneous servers according to their capabilities.

4. ROAR: RENDEZVOUS ON A RING

The problems above led us to develop a new continuous version of the sliding window algorithm that we call Rendezvous On A Ring (ROAR). Rather than simply arranging servers in a circular list, ROAR uses a continuous circular ID space $[0, 1]$. Each server is given a continuous range of this ID space that it is responsible for, such that all points on the ring are owned by some server. Thus ROAR uses the ring in a similar way to Chord [20], although that is where the similarity ends.

The basic idea is that given a partitioning level p, ROAR stores each object on the servers whose range intersects an arc of size $1/p$ on the ring (Figure 3); for searching, ROAR randomly chooses a starting point on the ring and forwards each query to p equally-spaced points around the ring. Whereas the basic sliding window algorithm stores a data item on exactly r consecutive nodes, ROAR stores on an arc of the ring in which, on average, there are r servers. This allows us to decouple query routing from the server identifiers. We now look at these mechanisms in greater detail.

4.1 Storing objects

Each data item is assigned a uniformly random identifier in $[0, 1]$. The data item now needs to be replicated on all the servers that are responsible for the ring segment of length $1/p$ that starts with the data item's ID. How this replication is actually done is independent of the basic functioning of ROAR. Possible strategies include:

- Push the data item to the first server, and then forward it from server to server around the ring.
- Have all the servers mount a shared filesystem such as GFS [14]. Servers periodically check the filesystem for files with IDs that should be stored in their range.
- Push the data item to all the relevant ring servers from a back-end update server that knows the ring topology.

A peer-to-peer solution using ROAR might use the first, whereas organizations with existing distributed filesystems might choose the second. Our implementation does the last of these, using a central coordinator to keep track of the ID ranges occupied by the servers.

4.2 Forwarding Queries

To perform a search, a query from a client is first sent to a front-end server. These front-end servers are responsible for partitioning the query and sending the sub-queries to p nodes on the ring. In our implementation, every front-end server is kept updated with the ranges of IDs on the ring for which each node is responsible.

The front-end server then picks a random ID q on the ring for this query, and sends sub-queries in parallel to the node responsible for ID q and the nodes responsible for IDs $q + 1/p,\ \ q + 2/p, \ldots, q + (p-1)/p$, modulo 1. As these IDs are $1/p$ apart on the ring and as each data item is replicated on a range of at least $1/p$, it is easy to see that the query will reach a node that holds every data item (refer to Figure 3). Each server that receives the query matches it against its data items and returns the matches (or the best matches if the query is for a very popular term) to the front-end server, which assembles the final list and returns it to the client.

The description above captures the basic idea of the ROAR algorithm, but not the whole story. The real benefit comes from an additional observation: if the front-end server chooses a partitioning value p_q for a query that is larger than p, the algorithm still matches all the data items. By default though, this would waste effort, as the query might hit more than one server that holds the same data item (as shown in Figure 4). However, if we embed the value p_q into the query, the servers can divide up the matching task by object ID so that no two servers match the same data item. To do this deterministically, a server that receives a query with logical destination id_{query} only runs the query against data items (objects) that satisfy the following two conditions:

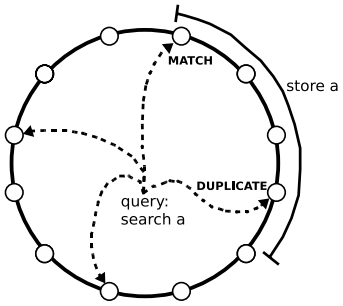


Figure 4: Duplicate matches are possible when $p_q > p$ is used. In this case, $r = 4$, $p = 3$ and $p_q = 4$.

$$id_{object} < id_{query} \tag{3}$$

$$id_{object} + 1/p_q \geq id_{query} \tag{4}$$

Data items that do not satisfy the second condition will be matched by the preceding server that received a sub-query (Figure 5(a)), while data items failing the first condition will be matched by the server receiving the following sub-query (Figure 5(b)).

There are two main reasons why it is so useful to be able to run queries with values of p greater than the bare minimum:

- Spreading a query across more nodes decreases latency. ROAR can dynamically trade off latency for total throughput (or if the nodes are not saturated, power consumption) without needing to first change the replication level.
- Allowing different values of p_q to be used for queries allows the partitioning to be changed while still serving queries.

4.3 Adding Nodes

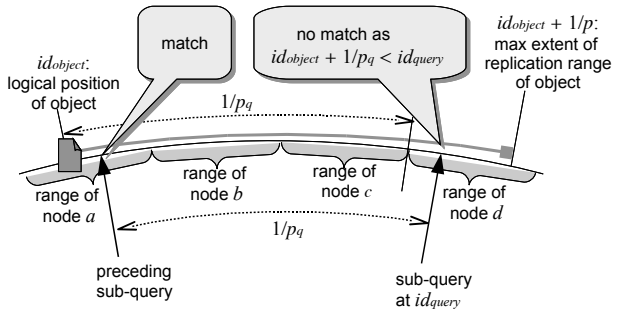
To function correctly, each server just needs to know its ID range. Typically, this must match up with the ranges of its immediate neighbors on the ring.

When a server joins the overlay, it is inserted between two other servers on the ring. The query load seen by a server is directly proportional to the fraction of the ring it is responsible for. Thus a simple strategy for inserting nodes is to pick the most heavily loaded node, and insert the new node as its neighbor.

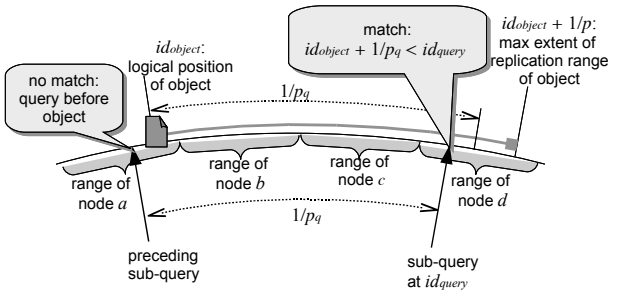
To start with, the new node has an infinitely small range, and so does not yet receive any queries. The nodes begins by replicating all the data items that traverse its ID. This download could be from its neighbor, but more likely it will be from a back-end filesystem to avoid putting extra load on an already loaded server.

Once the data download has finished, the new node communicates directly with its two neighbors to determine which of them is most loaded. It now starts to grow its range into that of the most loaded neighbor, requesting additional data items that overlap the range as it grows. Every few seconds it updates the front end servers with its new range, and also updates its neighbor so that the neighbor can drop data items in the overlapping range.

As the new node's range grows, its load will start to increase. Once the new node's load starts to approach that of its neighbors,



(a) Match by first sub-query



(b) Match by second sub-query

Figure 5: Avoiding duplicate matching in ROAR.

the rate of replication is slowed to a low background rate. In fact, nodes *always* compare load with their neighbors and expand their range very slowly into that of a more loaded neighbor. In this way, the nodes progressively distribute themselves around the ring, not with equal ranges, but with ranges that are the correct size to balance the load on the nodes, even if the nodes have heterogeneous processing power.

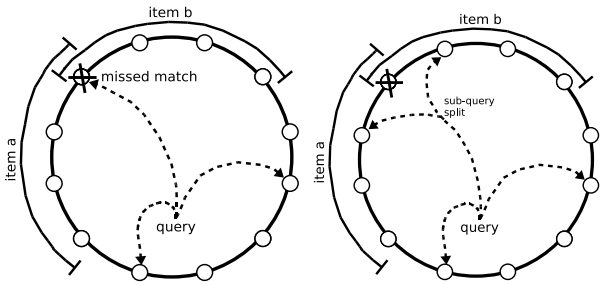
4.4 Removing Nodes

A node can be removed from the ring in a controlled manner by informing its neighbors that its load is now infinite. The two neighbors will grow their ranges into the range of the node to be removed by downloading the additional data needed. This data is typically a small fraction of the data a node already has: only $1/n^{th}$ of the data on a node starts or finishes at that node; it is this data that the neighbor will not already have. A neighbor of a shut-down node will need to download $1/2n^{th}$ of the data on average, if it takes over half of the neighbor's range and ranges are equal before the node is removed.

The query load will increase by as much 50% on the neighbors of the node being shut down, as their range has increased by 50%. However, in practice the neighbors' neighbors will expand their ranges as they see the load start to increase, so this upper bound is not normally reached.

What happens though if a node fails without warning? The failure will be discovered very quickly by the front-end servers, so they know not to route any more queries to it. However, we still want to match the data-items the failed node would have answered.

The front end server avoids starting a query on a failed node, but it ignores other failures when deciding the starting point. When it needs to send a query to a failed node, it uses a fall-back strategy. Each data item was replicated over an average of r servers that span a range of $1/p$; any of these servers could match the query instead of the failed node. We need to split the sub-query that would have been sent to the failed node in two because some data items' range might have ended on the failed node and some might have started



(a) A failure causing a missed match. (b) Failure-handling in ROAR.

Figure 6: A node failure can cause a query to miss a match. ROAR prevents this by splitting the failed node's sub-query in two and sending these to its predecessor and successor nodes.

on the failed node, as in Fig. 6(a). So long as we send the sub-query to two nodes, one before and one after the failed node, and so long as these nodes are not more than $1/p$ apart, then we are sure to match every data item that the failed node could match (Fig. 6(b)).

To spread the extra load across the maximum number of nodes we choose a pair of new targets for the sub-query as follows:

1. Let $fail_{lo}$ be the lowest ID held by the failed node and $fail_{hi}$ be the highest ID held by the failed node.

2. Pick a new first sub-query target id_{q1} randomly such that: $fail_{hi} - (1/p - \delta) \ < \ id_{q1} \ < \ fail_{lo}$.
δ is a small value that captures any uncertainty in the value of $1/p$. It is chosen so that $1/p - \delta$ is guaranteed to be less than $1/p_{old}$ for all recently used values of p_{old}.

3. Choose a new second sub-query target id_{q2} such that: $id_{q2} = id_{q1} + (1/p - \delta)$
This ensures the new sub-queries are never so far apart that a data item's range can fall between them and be missed.

4. Send both new sub-queries, but in the query request specify the original query ID. This is so that the only data items to be matched are those that the failed node would have matched, avoiding overlap with other sub-queries. Additionally, because the two new subqueries are maximally separated by almost $1/p$, their data sets are maximally disjoint, so they will produce very few duplicate matches.

The overall effect is that immediately after a node has failed and before any node has had a chance to download any failed items, all the queries are still being responded to correctly. The number of sub-queries being sent has increased by a fraction of $1/n$ because one extra query is needed for those queries that would have hit the failed node. The total matching load does not increase as nodes do not duplicate each other's work, but approximately $2n/p$ nodes share the extra $1/n^{th}$ of the load, so their load temporarily increases by a fraction of $2/p$.

The same algorithm applies for multiple failed nodes, but if either of the new sub-queries hits a second failed node, the process is simply repeated from step (2), choosing a new random value.

4.5 Changing the Replication Level

So far we have seen that for a given replication level r, we can partition queries for varying values of p_q, so long as $p_q \cdot r \geq n$. However, if, in an attempt to keep query latency low we are consistently running with values of p_q significantly larger than the minimum needed, then it does not make sense to keep sending all the

updates to all the nodes. Maintaining a replication level higher than needed requires extra bandwidth, using CPU and network capacity that could have been used to serve queries. Instead, we want to repartition by reducing r, hence increasing the minimum p.

If p is increased and r decreased, all the ROAR nodes have to do is drop a few objects from their local store. As it is always safe to run queries with higher p_q than needed, the front-end servers can just switch to the new p_q immediately, and let the ROAR nodes discard data in their own time.

Conversely, a ROAR system may discover that it is running with $p_q \cdot r = n$, using the minimum currently-available partitioning level. If the query latency is well below threshold, then p is probably too large, costing CPU cycles and hence increasing energy requirements[2]. If p is really excessive, nodes will saturate, and query delay will rapidly increase.

To decrease p to p', r must increase, and this is done by replicating each object $1/p - 1/p'$ further round the ring. The ROAR servers need to download the required objects from the filesystem, which can take some time. Further, the nodes will not all complete the download simultaneously. For correctness, when decreasing p to p', the front-end servers continue to partition queries p ways until they receive positive confirmation that every one of the ROAR nodes has obtained all the extra data needed. Only then do they switch to partitioning queries p' ways.

4.6 Load Balancing

The mean query rate seen by a node is directly proportional its range. To balance load, ROAR uses a slow background process in which each node extends its range into that of a more loaded neighbor. The goal is not to even out ranges, but to even out load so that a node's range is in accordance with its processing power.

If ROAR indexes N items in total, the number that need to be stored on a node i with a range of size g_i is the number of items that intersect the start of the node's range plus the number of items that start within the node's range; this is $N/p + N \cdot g_i$. On average $1/p = r\bar{g}$, so for sensible values of r, the N/p term dominates, and the amount of data *stored* by each node is fairly even between nodes, even if their ranges vary considerably.

However, although the mean query rate at a node depends on g_i, by choosing a random starting point on the ring for a query, we subject ourselves to the normal statistical variations associated with random processes. When we implemented ROAR it became clear that these variations could adversely affect load balancing sufficiently to impact query delay.

To greatly reduce this effect, we make use of "the power of two choices"[17]. When a front-end server partitions a query, it chooses two IDs at random on the ring and computes the expected delays in each resulting configuration of p servers. It will then choose the configuration that finishes first. To do so, the front-end maintains statistics about each server's processing power and RTT, as well as the tasks that have been assigned to that server and have not completed. To compute the expected finish time on a given server, the frontend simply uses $RTT + size/CPU$, if the server is idle; if not, it also takes into account the finish times of the existing tasks.

If servers are heterogeneous, there may be some sub-queries in the chosen configuration that finish much later than the rest, negatively impacting query delay. In this case, ROAR implements an optional load balancing mechanism at the frontend: before it sends the sub-queries, the frontend checks if the slowest server is expected to be more than 100ms behind the fastest one; if so the front-end uses a very similar mechanism to that described for handling failures. It splits the sub-query it expects to be slow in two, and reschedules the new subqueries. This continues until the difference in delay is below threshold, or the effective p_q reaches a predefined limit.

This mechanism complements the range load balancing mechanism, as it functions on a much shorter timescale: it can reduce delays even if ranges are not assigned according to processing power, at the extra cost of increasing p_q.

5. EXPERIMENTAL EVALUATION

To evaluate ROAR we built a prototype application and deployed it on 47 servers in the HEN testbed at UCL and on 1000 servers on Amazon's EC2 [1]. We also simulated ROAR extensively to examine scalability, but simulation fails to capture issues such as context switch overhead and I/O bottlenecks that impact real-world performance, so all the results below are from our testbed deployments.

The evaluation has two major goals. First, we wish to see how p impacts the properties of the system, including the average query delay, throughput, and system load. This gives insight into the range of values that are appropriate for p in practice, and tell us whether changing p has any sizable impact.

Second, we wish to evaluate ROAR. How does throughput and query delay scale with the number of nodes involved in the search? How easy is it to change p at runtime? How does ROAR cope with failures? How well do the load balancing mechanisms work?

5.1 The Application

Ideally we would have liked to evaluate ROAR using a full-blown web search application distributed across thousands of servers, as this is the most widely used distributed rendezvous application.

Unsurprisingly though, such large-scale search engines are not freely available for experimentation. We considered implementing a miniature search engine, but at small scale the query setup costs tend to dominate the query times, so the results would not be so meaningful. In the end we decided that to run a small scale experiment but still see meaningful results, we needed a more difficult matching application, where the matching costs would be comparatively large. Such an application still benefits significantly from being parallelized on the scales we can achieve on our testbed.

The application we chose to stress ROAR is called Privacy Preserving Search (PPS). Our system allows untrusted servers to match encrypted queries against encrypted metadata. The servers only learn the outcome of the match, not the contents of the query or the metadata [19]. It can be used, for instance, to protect privacy in online storage, such as Google Docs [2]. This application is CPU intensive because of the cryptographic operations required to perform a match. Files are encrypted before being stored, and encrypted metadata is also created and stored on the servers to allow the searching. When the user wants to retrieve some files, PPS runs queries to find which files the user is interested in. These queries are run on the servers, so the client platform can be extremely lightweight, such as a mobile phone.

In PPS, users each have many files (perhaps on the order of millions) for which they provide searchable metadata, and PPS's job is to answer queries for that data. To create metadata for our tests we used the files from a Linux filesystem. The test queries used randomly chosen keywords. From a usability point of view, we impose a delay bound of one second that the PPS system must meet.

We have two versions of PPS that exhibit different fixed costs. PPS is written in Java, and the cost of running the Java garbage collector is not negligible. PPS_LM (low memory) forces a run of the garbage collector immediately after finishing a query. This has

[2]The reader may think that the effect is negligible, but the temperature in our air-conditioned machine room ran 4°C hotter when our 47 ROAR nodes were fully loaded than when they are idling. We have since upgraded our A/C system.

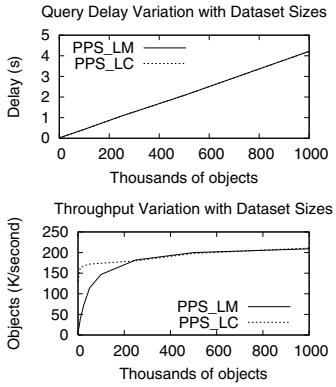


Figure 7: Single server performance

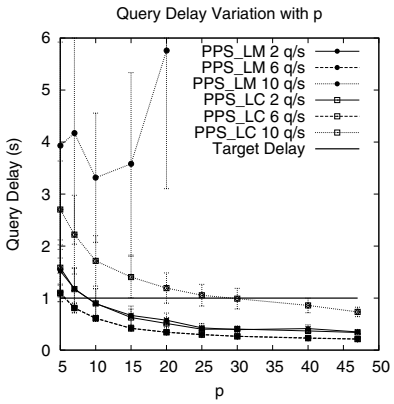


Figure 8: Effect of p on query delay

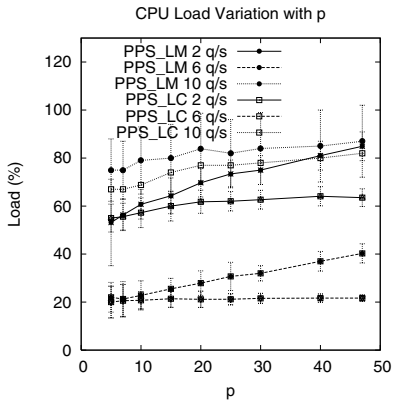


Figure 9: Effect of p on CPU Load

the advantages of minimizing memory usage and preventing the garbage collector running during a query, which would increase query delay, but the disadvantage of adding to the fixed costs of a query. PPS_LC (low CPU) does not force a garbage collection run after a query; it has lower fixed costs, but uses more memory and may exhibit more variable query delays.

We do not claim that PPS is an "optimal" application in any way, but merely note that real-world search applications also vary considerably in their ratio of fixed to variable costs. For example, Google's web search runs from memory, and has relatively low fixed costs because all users search the same web index. In contrast, with Google's Gmail, queries from different users obviously have to search different indexes. It doesn't make sense to store all such indexes in memory for all users. Loading a file from disk has a large seek/rotate latency followed by a fast consecutive read phase, so has a comparatively high fixed cost.

As a test application PPS shares the main properties with web search. The mechanisms are different, but the average cost of matching in both cases has a large component that grows linearly with the number of documents searched, although PPS search costs are less dependent on the contents of the query. Both applications are bottlenecked on CPU cycles or memory bandwidth. The different versions of PPS have quite different fixed costs, as we would also expect when comparing regular web search with webmail search.

5.1.1 *Characterizing PPS*

Before applying ROAR to PPS, we first examine how PPS performs on a single machine. The main issue is query delay as shown in Fig. 7. As expected, once the fixed costs are satisfied, query delay increases with the number of metadata objects to be searched.

When the number of objects is smaller, the fixed costs associated with running a query cease to be negligible, which shows up as a performance drop off in the bottom graph in Fig. 7. The drop is steeper for the low memory version.

In absolute terms, when searching one million metadata items a single server takes 4.2 seconds to perform a query, which is unacceptable, especially if many users are using PPS simultaneously. Ideally we would like a PPS system that is able to respond to multiple, simultaneous requests in at most one second each. We will now use ROAR to distribute PPS across multiple networked servers in order to achieve this aim while increasing request throughput.

5.2 Basic Tradeoff

To examine how p impacts query delay and throughput we created a dataset of one million files. From these we created an encrypted metadata index consisting of 30 keywords per file, plus some other metadata. We distributed this index to our 47-server ROAR deployment, and searched it with queries consisting of two randomly chosen keywords that must both match for the file to match. While this is a slightly artificial workload, the precise contents being searched are not terribly relevant as distributed rendezvous is content-agnostic.

To allow a single server to search its part of the index in one second, we started with a value of $p = 5$, the smallest value that has any hope of meeting our target search latency. From here, we progressively increased p all the way up to the largest possible value of 47, at which point every server is processing 1/47 of every request. For each value of p, we tried workloads from two queries per second up to ten queries per second; these corresponded to light, moderate, and heavy workloads.

5.2.1 *Query Latencies Decrease with p*

The query latencies are shown in Figure 8. At low and moderate load, query latency scales inversely proportional to p, as we would hope, and is similar for both versions of PPS. It is clear that to achieve a target latency we need to have p greater than a particular threshold. However, this threshold is not fixed, but depends on the offered load. This should not be a surprise: a query cannot complete until all its sub-queries complete. There is inevitably some short-term variation in the loads on the different machines, so some sub-queries are delayed.

The heavy workload is sustainable at any p by the LC version, and shows a similar slope to the other workloads. However, average delay for LM decreases initially, then increases as $p = 20$. This is because nodes are close to saturation at this point, and any small variation in query arrivals induces longer delays. If we increase p further, LM saturates some nodes and cannot cope with the load. This example serves to show that fixed overheads decrease the maximum throughput when p increases.

5.2.2 *Query Overheads Increase with p*

Figure 9 plots mean CPU load (as measured by the "top" utility) for varying values of p and for each of the workloads. The error-bars show the standard deviation. The trend is clear: CPU utilization increases with p. For the low memory version, the curves show relative increases of 80% (from 22% to 40%), 54%(from 53% to 85%), for the workloads of two and six queries per second respectively. For the LC version, the relative increases are of approximately 10% in both cases. The differences between the two versions show the overhead of more frequent garbage collection.

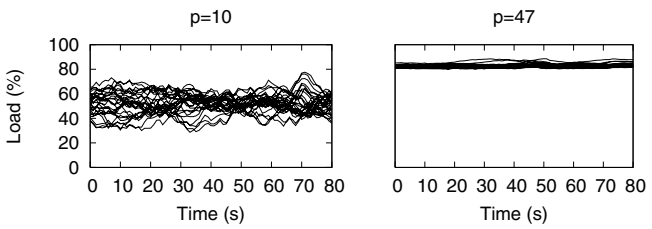


Figure 10: Average system load for each node

Model	PE 2950	PE 1950	PE 1850	Sun X4100
PPS_LM	51W	50W	10W	7W
PPS_LC	18.9W	17W	3W	2W

Table 1: Energy Savings running at $p = 5$ instead of $p = 47$

At the highest load, the increase is more modest for LM, because the nodes are saturated. For LC, the relative increase is 22% (from 67% to 82%).

To see this in more detail, Figure 10 shows a 20-second average of CPU load for all our PPS_LM servers when $p = 10$ and $p = 47$ with 6 queries per second. When $p = 10$, individual load fluctuates much more as queries come and go. When $p = 47$ there are few idle times and load is heavily and constant.

Our cluster can handle two of these workloads with any value of p, but using large p values uses enough extra CPU power to waste considerable energy (Table 1). Comparing $p = 5$ with $p = 47$, our newer servers[3] were measured to consume 18W more with PPS_LC and 50W more with PPS_LM. Our older servers[4] have less good CPU power management, so less savings. We expect that the latest Intel Nehalem CPUs will show even greater savings than those shown.

Each query requires a disk seek then reading 250MB of contiguous data. On our systems the kernel disk buffer cache reduces I/O and PPS is largely CPU bound, but in machines with less memory disk performance might matter. The ratio of seeks to reads increases with p, wasting I/O bandwidth. The Maxtor 10K V disks in our servers take 7.5ms on average to seek and transfer data at 73MB/s. When $p = 5$ it takes each server 680ms to sequentially read its part of the data; when $p = 47$ it takes 80ms. At this point seeks accounts for 10% of the transfer times, so if the system were disk bound, using a higher p would reduce maximum throughput by 10%.

Finally, the bandwidth required to run a single query increases proportionally[5] with p. This does not create a sizeable impact on energy consumption, but will increase usage of the scarce cross-section bandwidth. We will go in more detail on cross-section bandwidth usage in Section 5.6.

In summary, increasing p above the minimum needed to satisfy the required delay bounds increases system load. Depending on the workload, very large values of p may reduce the peak throughput that can be handled, or at the very least waste resources and energy.

5.2.3 *Update Overhead increases with r*

To see how server throughput (matches/second) is affected by background updates of the dataset we created medium (5K updates/sec) and high (20K updates/sec) update rates. Figure 11 shows

[3]Dell PowerEdge 2950, with two quad-core Xeon CPUss and PowerEdge 1950, with two dual core Xeon CPUs

[4]Sun X4100 with one AMD Opteron CPU and Dell 1850 with one older Xeon CPU

[5]In our PPS deployment the increase is modest: from 2.5KB to 24KB per query

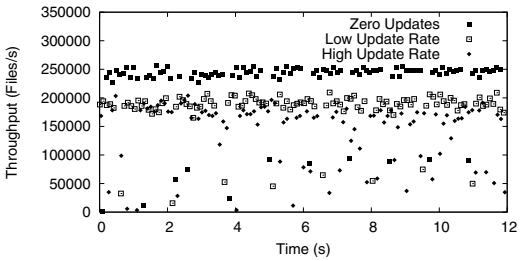


Figure 11: Effect of updates on server throughput

a single server's throughput in these conditions in comparison with no update load. Unsurprisingly, the higher the load the bigger the reduction in throughput. For the moderate load, the average drop in throughput is 20%; for the higher load, the drop is even sharper. In applications like PPS, where the data are stored to disk, this effect needs to be considered when determining and changing r.

5.2.4 *Does the trade-off matter?*

We have seen that larger values of p give lower delays but higher system load, so there is a natural push of p to the minimum value that achieves the desired query latency. We've also seen that higher update rates, which can result from larger values of r, reduce server processing speed; thus there is a push to minimize r. Taking these two together, it follows that a distributed rendezvous system should be run close to the minimum combination of p and r, that is $p \cdot r = n$, where n is the server count.

To summarize, minimizing p subject to latency constraints seems a sensible goal. However, small p implies large r, which, in turn, increases the bandwidth used to replicate the changing dataset and the update processing load of the servers. Thus the ability to dynamically change the tradeoff between r and p is very useful to ensure that the system runs at a good near-optimal operating point.

5.3 Changing p Dynamically

One of the benefits of ROAR is its ability to repartition on-the-fly while still serving queries. To investigate how this works in practice we implemented a simple adaptive strategy to change p based on the average query latency seen by the front-end servers. Given an average target delay of one second, the front-end servers instructed the ROAR servers to adapt p to the minimum value that still yielded the target latency (allowing for an error of 10%). Increasing p had no cost, of course, but to decrease it servers needed to copy data; this increased their load, so is more interesting.

We ran an experiment with this adaptive strategy starting with no replication and $p = 40$, as if the system had just booted. We loaded the system with a moderate search rate of six queries per second, and plotted the behavior of the system as time goes by in Figure 12.

To start with, CPU load is very high and the query delay is less than it needs to be. We see that ROAR can quickly change p with minimal disruption to queries: within minutes average CPU load decreases while query delay stays within acceptable bounds.

This same experiment can serve as an example of adaptation for flash crowds: when load becomes too high (above some predefined threshold) the system sacrifices query latency for lower CPU load.

The strategy of minimizing p while maintaining the desired query delay seems sensible, yet in reality many other factors need to be taken into account. The cost of pushing dataset changes out to nodes gets higher as p decreases, so using larger values of p might be desirable. In addition, p might need to be increased to reduce the memory strain on each server (this seems to be a constraint in

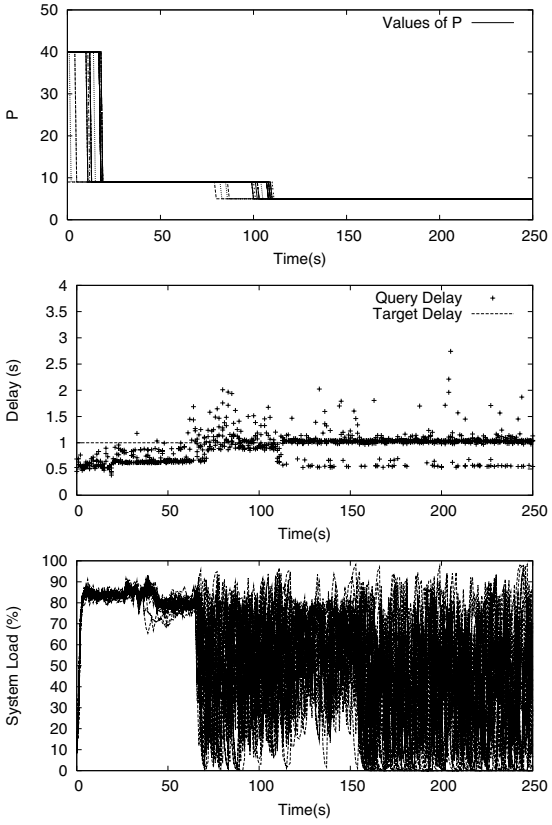


Figure 12: ROAR Changing p Dynamically

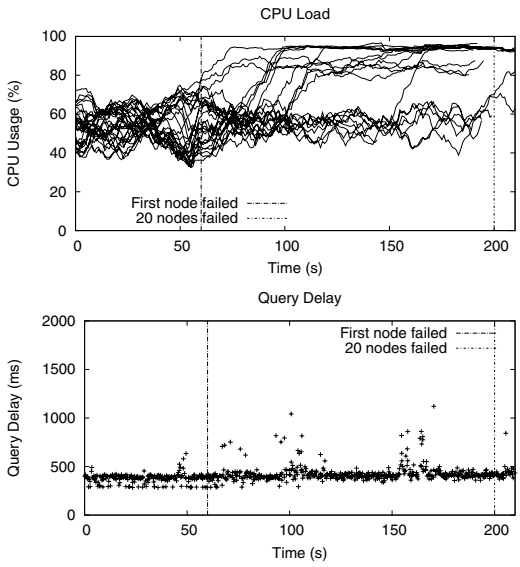


Figure 13: Effects of 20 Node Failures on ROAR

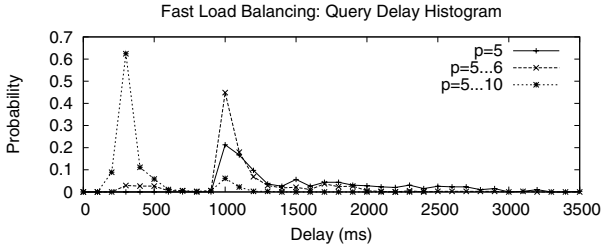


Figure 14: Delay Distribution with Fast Load Balancing when using $p_q > p$

Google's case). Bandwidth utilization depends on p too. In some cases rather complex optimization functions might be required; in any event, a ROAR system can implement the required changes of p so long as an optimization function can be defined that captures the relevant constraints.

5.4 Node Failures

What is the impact of server failures on ROAR? We are more interested in short term effects, as in the long run the load balancing mechanism evens out load across all the servers (see next section).

To test the impact, we set $p = 20$, so that r was very small (approximately 2). This reduces ROAR's options for alternative servers to the bare minimum, and hence represents a worst case for the increase in load on the remaining nodes caused by a node failure. With this setup, we ran queries at a rate of six per second, then killed a single server. Query delays remained roughly the same. We noticed a small increase in CPU load of roughly 10% for the two neighbors of the failed node. This agrees with our analytic predictions in Section 4.4.

In the second experiment we generated queries at a lower rate (3 per second) and progressively killed 20 out of the 47 servers. To maintain correctness, we did not kill consecutive servers because with such an artificially small value of r there was not much redundancy. The effect on query delay and server CPU load is plotted in Figure 13. The average CPU load doubles for most servers, as expected, though query delays only increase marginally for this workload. Clearly if the initial workload had been higher than 50%, this failure would have pushed load above 100% and so query delays would have been affected. In such a scenario the correct course of action would then be to decrease p, as shown in Section 5.3.

To summarize, the results show that ROAR handles node failures gracefully, and so long as the load does not exceed 100%, query execution is not disrupted.

5.5 Load Balancing

The previous experiments were conducted with homogeneous servers. In a data center it is unlikely that all servers will be equally fast, as machines are bought in batches and computing power increases from one batch to another. To test this effect, we included 15 powerful machines in our testbed (each server with two quad-core processors). These run the same million metadata query four times faster than our slower servers.

To cope with heterogeneous servers, ROAR implements two load balancing mechanisms(§4.6):

- The background process by which ranges migrate.
- A request scheduling mechanism implemented in the front-end load balancer.

These run simultaneously, though on different timescales.

The front-end load balancer was not enabled in any of the experiments up to this point, but with heterogeneous servers it helps significantly. We started all the servers, assigned them equal ranges, set $p = 5$ ($r \simeq 9$), and generated six queries per second. Figure 14 shows the distribution of delays when the front-end load balancer is turned off ($p = 5$), when it is allowed one extra subquery ($p = 5...6$), and when it is allowed to increase p_q as high as

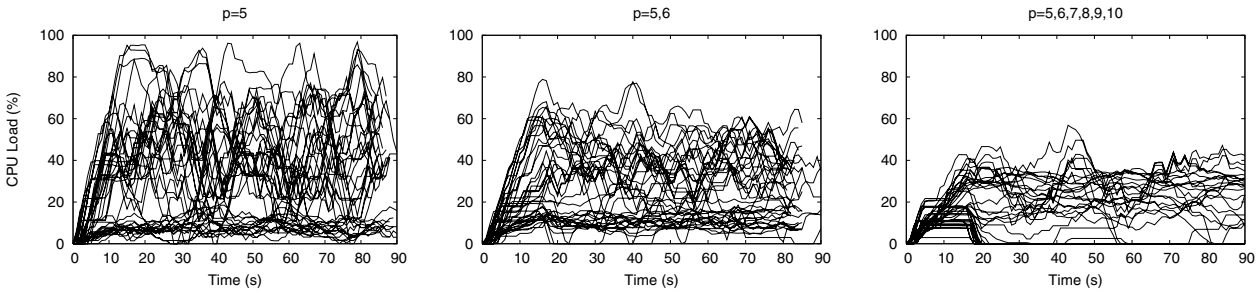


Figure 15: Fast Load Balancing with $p_q > p$

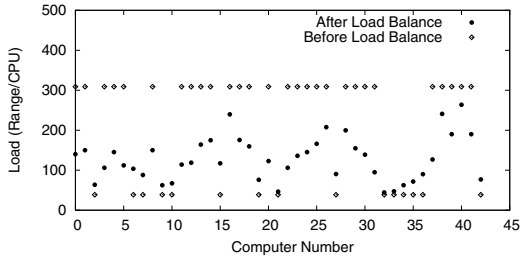


Figure 16: Range Load Balancing

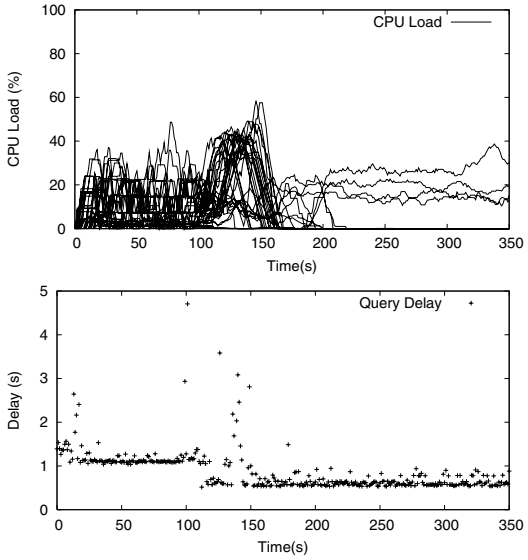


Figure 17: Effects of Range Load Balancing

10 if needed. It is clear that this mechanism is effective at moving load onto the faster servers.

Figure 15 shows the load on the machines as the load balancer learns which machines are fastest. In the $p = 5$ graph, we can see a band in CPU load at around 12.5%; this corresponds to the fast servers which are given similar workload to the slower servers. As p_q is allowed to increase, this band moves up, and the upper band (the slow servers) moves down. When p_q is allowed to grow up to 10, sometimes slow servers are not given any work, simply because all the load can be processed quicker on the fast servers. When the load is increased, the slow servers start to be used again.

To test the long-term range load balancing, we started the servers with equal ranges and ran one query per second. Before long the front-end servers compute a new configuration for the network where ranges are better balanced. The load balancing procedure iterates many times, evening out ranges between neighbors where the load difference is greater then 1.5.

The results are encouraging: the big range differences between neighbors are amortized (Fig. 16). The zig-zag shape of the resulting load allocation is the effect of the distributed, neighbor-only load balancing mechanism. The effects of load balancing are clear in Fig. 17. This range expansion increases the effectiveness of the front-end balancer: for light loads most servers are not used at all, as the powerful servers can run all the queries in less time.

Many of these unused servers can actually be put to sleep to save electricity. They do however need to be updated when they are woken again. One strategy is to wake some of them periodically for updates to reduce the wake up time when they are actually needed.

5.6 Cross-Sectional Bandwidth Usage

Typical data-center networking architectures connect racks of servers with one switch per rack, and have one or two layers of switches that interconnect the racks. The tree hierarchy causes bandwidth further up in the tree to be scarce compared to intra-rack bandwidth. Although it is possible to increase the cross-sectional bandwidth, achieving full bandwidth between any two nodes is very expensive. As a consequence, cross-sectional bandwidth usage is a major concern in data-center algorithm design. In this context, it becomes important to understand how ROAR compares with simple partioning in cross-sectional bandwidth usage.

Distributed algorithms can exploit the physical network structure to minimize cross-sectional bandwidth usage. We can either place data replicas in a small number of racks, or attempt to run a query in as few racks as possible. These are mutually exclusive at large scale, so we can typically optimize cross-sectional bandwidth usage for only one of the two.

Google's web search generates too little cross-sectional bandwidth to be of concern, mostly because it has many data-centers worldwide [9]. However, if all the servers were in the same data-center, would this be an issue?

Our estimates of bandwidth usage[6] for running queries at Google give 1Gbps to serve 1000 queries per second, with $p = 1000$. This assumes no caching; in reality, cache hit ratios can be on the order of 30-60% [10] so the bandwidth overhead is much smaller: 500Mbps. This is modest in comparison with updates to the index: if r is 80, and the entire 10TB index is redistributed daily[8] to 80 replicas, the total bandwidth needed is around 75*Gbps*. Even if

[6]The query keywords and options take 50 bytes at most; the reply is 80 bytes long if it includes 10 64 bit document IDs. Altogether, this takes 130 bytes.

p	100	250	500	1000
Delay (ms)	997	341	1132	2183
CPU Usage	10%	12%	15%	19%
Match Delay (ms)	430	160	80	20
Match Variability	1.2	1.5	2.5	4
Schedule Delay (ms)	1.17	3.4	9.2	23
Serialize Delay (ms)	8.3	24	50	155

Table 2: ROAR performance running on 1000 servers in EC2

only incremental updates are sent, it would make sense to optimize update bandwidth.

Google could place one cluster of nodes (i.e. nodes with the same data) in as few racks as possible, say l. To update the data, each item needs to be sent to a single machine in each rack; it will be then propagated locally within the rack. Assuming incoming update traffic is D, updates will use lD cross-sectional bandwidth; in the example above, and assuming 40 servers per rack, this could result in 2Gbps of cross-sectional traffic.

ROAR can similarly use physical placement of servers to minimize update cost, by assigning servers in the same rack to be consecutive on the ring. In this case, each update will be pushed to l or $(l+1)$ racks. ROAR will generate $(l+1)D$ cross-sectional traffic for each update, which is marginally more than Google.

To implement this optimization in ROAR, it suffices to use the peer-to-peer like update algorithm we have described: the updates for an object are pushed to the server responsible for that object's ID. This server forwards to its successor, and so forth, as long as the successor is within the replication range. Almost all of these hops will be intra-rack.

5.7 Large Scale Deployment

Small-scale tests on our testbed show that ROAR works, but we also wish to see how it scales. ROAR stores r replicas of each data item, and splits each query p ways while ensuring $p \cdot r = n$. This is the lower bound for *all* distributed rendezvous algorithms, so we are confident that ROAR's basic costs scale well. Simulation indicates that the algorithms should scale, but there are always practical surprises when scaling a system up significantly. Our immediate concern is the frontend scheduler, which is centralized.

We briefly acquired a thousand servers from Amazon EC2 [1]. These are virtualized servers, each with a 1.7Ghz CPU and 1.7GB of memory, plus a large local hard drive. Our front-end server is instantiated on a more powerful machine with eight virtual processors and 17GB of memory.

Basic performance of PPS on a single EC2 instance is roughly half that on our HEN servers because the CPU is slower: a query of one million metadata items takes eight seconds.

We created a larger dataset of 5 million entries, and replicated it at $r = 10$ on 1000 servers. We then ran one query per second at different p values (min p for correctness is 100). Table 2 summarizes the results. Query delay initially decreases as p goes from 100 to 250, but then increases after that. Average CPU utilization increases with p as we expect: it roughly doubles when p goes from 100 to 1000. As the CPUs are not overloaded, the u-shaped delay curve is intriguing.

We profiled the frontend server to see how local computation affects latency. Scheduling delay increases roughly with $n \log n$ and reaches 25ms on average when $n = 1000$. The time to compose and send the 500 byte query from the frontend application also increases with n: it takes 125ms on average to send a message to all the 1000 servers. Although not negligible these delays can be easily reduced in an optimized implementation and are not a scaling concern. They are not large enough to explain the u-shaped curve.

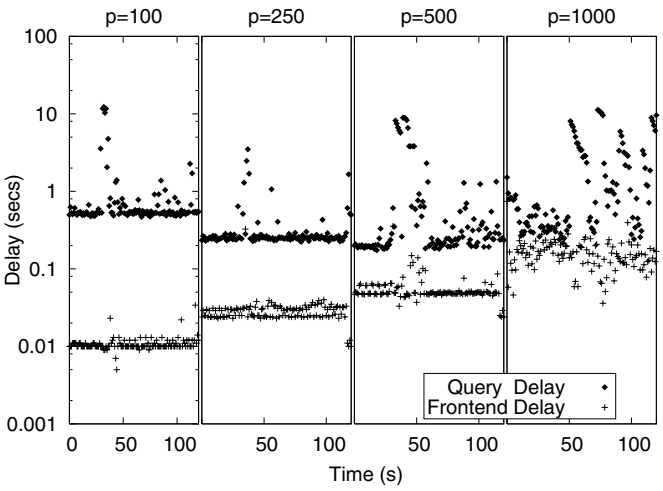


Figure 18: Delay Breakdown as seen at Frontend Server

We then examined the query matching times on the ROAR nodes. The mean performance is as expected: delays decrease with $1/p$. However, larger values of p exhibit higher variability in run-times: variability[7] increases from 1.2 to 4 when p goes from 100 to 1000.

To nail the cause of high delays observed, Figure 18 shows a real-time breakdown of frontend delays and query delays for various values of p. Many queries finish very quickly when $p = 1000$, just after all the data has been sent. Variable round-trip delays made us wonder if we were bottlenecked on bandwidth, despite the low transmit rate of 4Mb/s. Brief tests with iperf showed this was not the case, but they did reveal a mean drop probability of roughly 1 in 1000 packets, presumably caused by competing uses of EC2. As we use TCP between the frontend and each ROAR server, a drop on any flow delays the whole query. The query rate to each server is low, so TCP's fast retransmit cannot kick in and a lost packet has to wait for a TCP retransmit timeout. The large delays spikes in Figure 18 indicate losses are bursty too, making matters worse. A simple technique might mitigate these losses: the frontend should resend unfinished query parts as soon as most of the query has completed. As least for our application, this implies that UDP might be a more appropriate transport for ROAR.

Our large-scale deployment gives us confidence that ROAR itself scales well. It also provided insight in the effects of p, beyond the ones we observed in our small scale testbed. In particular, larger p values greatly exacerbates any inherent variability in runtimes, increasing overall query delays. This strengthens our belief that dynamically adapting p is advantageous.

6. RELATED WORK

There are many proposed distributed rendezvous solutions in the literature [5, 12, 24, 25, 13]; almost all offer a fixed trade-off between the partitioning and replication levels. The Google cluster architecture [5] is the classical cluster-based solution, with a fixed r-p trade-off.

Another solution is the Load Balancing Matrix (LBM) [13]. LBM is the only solution we are aware of that allows changing r dynamically. LBM maps clusters on a DHT: server i from cluster j is mapped to the server in charge of $hash(i, j)$. When repartitioning,

[7] defined as the ratio between the finish time of the slowest node and the average finish time of all nodes running a query.

LBM inherits most of the problems of the Google approach, but does not require changing the network structure. However, LBM has load balancing problems as virtual cluster servers are mapped using consistent hashing onto the Chord ring: with high probability, the busiest server will host $\log n / \log \log n$ cluster servers. To fix this, each server has to insert itself many times on the ring (as many as $\log n / \log \log n$), which significantly increases distributed rendezvous costs for large networks.

There are a few randomized solutions: Ferreira et al. [12] use random walks for both object storing and for queries, while BubbleStorm [25] uses bubbles to speed up object storing and query execution. These algorithms are built for peer to peer systems so have great resilience yet their operating costs are much higher (for instance with BubbleStorm $p \cdot r = 4n$).

Structured Overlays and P2P Search. Much research has gone into executing queries on structured overlays, including keyword search [23, 22] and range queries [7]. These solutions are applicable to many problems. However, when queries are complex, content distribution is skewed, or content is unavailable (as with encrypted search), content-based solutions do not work well. ROAR's content-agnostic approach is a better solution in many such cases.

Distributed Databases. Research in distributed databases aims to optimize execution of powerful relational queries in a distributed setting [21, 26, 16]. ROAR is much simpler: it is just a "select" operation executed in a distributed manner on a single table. In effect, ROAR can be used as a tool underlying traditional databases to optimize access to large tables with poor indexing options. At a conceptual level, ROAR is similar to the exchange operator proposed by Graefe et al. to provide extensible query execution [15].

Distributed Computation. There are many algorithms for distributing computation among machines [6, 4, 11]. Google's MapReduce [11] offers a simplified, functional programming model that hides parallelization from the programmer. ROAR offers a weaker programming abstraction, equivalent to the "map" operation, but differs in its handling of data objects: while MapReduce moves data to the servers performing the computation, ROAR will run the computation on enough servers such that all the data objects are visited without actually moving the data objects. Instead, ROAR allows the application to change r, which controls the minimum number of servers that must be visited. Not copying data for every query allows ROAR to save bandwidth and obtain smaller delays.

7. CONCLUSIONS

The performance of web search engines is heavily influenced by the partitioning level p, which controls how an ensemble of servers handle queries and store a web index. This parameter is the primary control that determines search latency, and so has a huge impact on the usability of distributed search systems. Despite this and the fact that p should be continuosuly adapted according to the system's load in order to achieve optimal performance, search engines such as Google rely on simple distributed rendezvous algorithms that do not allow for dynamic reconfiguration of p.

In this paper we introduced ROAR, a novel distributed rendezvous algorithm that allows on-the-fly re-configuration of p at minimal cost while still servicing queries. Further, ROAR can add and remove servers without stopping the system, cope with temporary and permanent server failures, and provide very good load-balancing even in the face of servers having heterogeneous hardware capabilities.

We have provided experimental results that support these claims and that show that the ability to change partitioning dynamically has many benefits, from allowing the network to cope with load fluctuations gracefully to reducing bandwidth and energy costs. We derived these results by implementing a privacy-preserving search application that used ROAR as its underlying algorithm, and running experiments on a 47-server dedicated testbed and on a 1000-server configuration using Amazon's EC2. Our experiments show that ROAR works well in practice: it can cope with failures and it balances load well. Given a target query delay, ROAR can automatically reconfigure the network to achieve that delay while minimizing other costs.

In the future, we hope to test ROAR more on large clusters with thousands of nodes using a more robust transport, build smarter optimization criteria, and to see how ROAR can be used in other search applications.

8. REFERENCES

[1] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[2] Google Docs. http://docs.google.com/.

[3] The Google Search Query - a technical look. http://www.webmasterworld.com/google/3694079.htm.

[4] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster i/o with river: making the fast case common. In *Proc. Workshop on I/O in parallel and distributed systems*, 1999.

[5] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *Micro, IEEE*, 23, 2003.

[6] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control a batch-aware distributed file system. In *NSDI*, 2004.

[7] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 34(4), 2004.

[8] M. Cutts. Gadgets, Google, and SEO: Explaining algorithm updates and data refreshes, Dec. 2006.

[9] J. Dean. Personal Communication. Google.

[10] J. Dean. Challenges in Building Large Scale Information Systems. Keynote Presentation at ACM WSDM, 2009.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[12] R. A. Ferreira, M. K. Ramanathan, A. Awan, A. Grama, and S. Jagannathan. Search with probabilistic guarantees in unstructured peer-to-peer networks. In *Proc. P2P*, 2005.

[13] J. Gao and P. Steenkiste. Design and evaluation of a distributed scalable content discovery system. *IEEE JSAC*, 22, Jan. 2004.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[15] G. Graefe and D. L. Davison. Encapsulation of parallelism and architecture-independence in extensible database query execution. *IEEE Trans. Softw. Eng.*, 19(8), 1993.

[16] B. Kröll and P. Widmayer. Distributing a search tree among a growing number of processors. *SIGMOD Rec.*, 23(2), 1994.

[17] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, 2001.

[18] N. Patel. Learning from google's data centers. http://www.pronetadvertising.com/, 2006.

[19] C. Raiciu and D. S. Rosenblum. Enabling confidentiality in content-based publish/subscribe infrastructures. In *Proc. Securecomm*, 2006.

[20] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. SIGCOMM*, 2001.

[21] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: a wide-area distributed database system. *The VLDB Journal*, 5(1), 1996.

[22] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proc. Sigcomm*, 2003.

[23] C. Tang, Z. Xu, and M. Mahalingam. psearch: information retrieval in structured overlays. *SIGCOMM Comput. Commun. Rev.*, 33(1), 2003.

[24] W. W. Terpstra, S. Behnel, L. Fiege, J. Kangasharju, and A. Buchmann. Bit zipper Rendezvous—Optimal data placement for general P2P queries. In *Proc. EDBT Workshop on Peer-to-Peer Computing and DataBases*, 2004.

[25] W. W. Terpstra, J. Kangasharju, C. Leng, and A. P. Buchmann. Bubblestorm: resilient, probabilistic, and exhaustive peer-to-peer search. In *Proc. SIGCOMM*, 2007.

[26] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *Proc. VLDB*, 2003.