ROAR: Increasing the Flexibility and Performance of Distributed Search

Costin Raiciu

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

UCL.

Department of Computer Science

University College London

April 11, 2011

To Andrei, Cristina and my parents

Abstract

Search engines are a fundamental building block of the web. Be they general purpose web search engines, product search engines for online catalogues or people search in online networks, search engines provide easy access to a huge amount of information. To cope with large amounts of information, search engines use many distributed servers to perform their functionality.

For instance, 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 search algorithms is the trade-off between replication and partitioning: increasing the partitioning level typically improves query completion time since more servers handle the query. However, partitioning too much also has drawbacks: startup costs for each sub-query are not negligible, and will decrease total throughput. Finding the right operating point and adapting to it can significantly improve performance and reduce costs.

In this thesis we propose that the tradeoff between partitioning and replication should be easily configurable. To this end 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 experimentally show that it is possible to dynamically adjust the partitioning level to cope with different loads while meeting target query delays, and in doing so the system can reduce its power consumption significantly.

To test ROAR we introduce Privacy Preserving Search: a particular search application that allows users to store encrypted data online while being able to easily search that data. Our contributions include novel protocols that allow PPS for numeric values, as well as a proof of concept implementation of PPS running on top of ROAR and allowing users to match as many as 5 million files in well under 1s.

Acknowledgements

My PhD has brought me to the highs of exultation infused by fresh ideas and the lows of experiments gone the wrong way. It was a journey I would always remember, and one I would always start again if I could. I met, worked with, and made friends with many people during these years, and here I'd like to express my gratitude to all of them for helping me become a researcher.

My advisor David Rosenblum, always calm and collected, gave me the chance to study at UCL, and helped me endure the many paper rejections and find a way to publish what was obviously a good idea - ROAR, the basis of my PhD :)! Mark Handley, my second advisor and later my boss during the Trilogy project has showed me what good research is, and introduced me to my now major area of interest - computer networking.

My colleagues and friends Felipe Huici and Adam Greenhalgh from the nets group helped me steer my way around Hen, and provided invaluable help when I needed it most - for paper deadlines. Petr Marchenko and Andrea Bittau helped me unwind with regular fussball matches. My friend Mo always put a smile on my face, even when I was really down. Paper discussions with people in the nets group including faculty members Brad Karp, Damon Wischik and Kyle Jamieson, as well as colleagues Piers O' Hanlon, Georgios Nikolaidis helped me find interesting problems and compelling solutions in the vast systems literature.

A big thanks goes to all my friends in the SSE group, where I spent the initial three years of my PhD: Clovis Chapman, Genaina Rodrigues, Leticia Duboc, Mirco Musolesi, Stephanos Zachariadis, Andy Maule, Danielle Quercia, Michelle Sama and many others.

Nithin Umapathi, a great friend I found in London, broke the monothony of research with intriguing discussions about the world over coffee. His views of the world made a lasting impression on me.

A great thank to my parents that gave me a chance to do a PhD, by bringing me up to love school and science, and who supported my throughout my PhD. Without their help I wouldn't be in London finishing the PhD now.

Cristina was close to me during my highs and lows, and gave me enormous support throughout. It is great to be with you! Our son Andrei, born during my PhD, made me want to leave home late and come back early. He always puts a smile on my face and makes me feel fulfilled. I thank them both.

Contents

1 Introduction **11**

2 Problem Space **15**

2.1 Problem Definition 17

2.1.1 Running a Query 19

2.2 The Distributed Rendezvous Trade-off 20

2.3 Scope 20

2.3.1 Server Reliability 21

2.3.2 Communication Costs 21

2.3.3 Application Delay Bounds 22

3 Solution Space **23**

3.1 Partitioned Distributed Rendezvous 24

3.2 Randomized Distributed Rendezvous 25

3.3 Sliding Window Distributed Rendezvous 26

3.4 Limitations of Existing Solutions 28

4 ROAR: Rendezvous On A Ring **29**

4.1 Storing objects 30

4.2 Forwarding Queries 30

4.3 Adding Nodes 32

4.4 Removing Nodes 32

4.5 Changing the Replication Level 34

4.6 Load Balancing: Proportional Ranges 35

4.7 Multiple Sliding Windows 36

4.8 Running Queries on Heterogeneous Servers 36

4.8.1 Scheduling Algorithm 37

4.8.2 Optimisations 39

4.8.3 Multiple Front-End Servers 41

4.8.4 Sending Queries Reliably 41

4.9 Managing Ring Membership 42

4.9.1 Adapting to Changing Load . . . 43
4.9.2 Reducing Cross-Sectional Bandwidth Usage . . . 44

5 Application: Privacy Preserving Search 45
5.1 Motivation . . . 45
5.1.1 Limitations of Online Privacy . . . 46
5.2 Basic Approach and Scope . . . 47
5.3 Analysis of the Index-Based Solution . . . 47
5.3.1 Bandwidth Comparison . . . 48
5.4 Definition of Privacy Preserving Search . . . 50
5.4.1 Security Preliminaries . . . 50
5.4.2 Security Assumptions and Scope . . . 51
5.4.3 Problem Definition . . . 51
5.4.4 Limitations of Confidentiality . . . 53
5.5 Solutions for Privacy Preserving Search . . . 54
5.5.1 Equality Matching . . . 55
5.5.2 Keyword Matching . . . 56
5.5.3 Numeric Matching . . . 59
5.5.4 Supporting Ranked Queries . . . 62
5.5.5 Supporting Generic Queries . . . 62
5.6 Implementation . . . 63
5.6.1 Overview . . . 63
5.6.2 Managing Metadata . . . 64
5.6.3 Running Queries . . . 64
5.6.4 Metadata Encoding . . . 65
5.6.5 Multi-Predicate Queries . . . 65
5.7 Evaluation . . . 66
5.7.1 Dynamic predicate ordering . . . 69
5.7.2 Query delays with varying numbers of metadata . . . 69
5.8 Related Work . . . 71
5.9 Conclusions . . . 72

6 Analytical Evaluation 73
6.1 Query Delay . . . 73
6.1.1 Bounding Optimal Query Delay . . . 74
6.1.2 Query Delay Comparison when $p_q = p$ . . . 75
6.1.3 Query Delay Comparison when $p_Q > p$ . . . 81
6.1.4 Analysis of ROAR Mechanisms . . . 83
6.2 Fault Tolerance . . . 84

6.3 Changing the p/r tradeoff . . . 86
6.4 Comparison Conclusions . . . 87

7 Experimental Evaluation 89
7.1 Experimental Setup . . . 89
7.2 The Application . . . 90
7.3 Basic Tradeoff . . . 91
7.3.1 Query Latencies Decrease with p . . . 92
7.3.2 Query Overheads Increase with p . . . 92
7.3.3 Higher Overheads=Wasted Resources . . . 93
7.3.4 Update Overhead Increases with r . . . 94
7.3.5 Does the trade-off matter? . . . 94
7.4 Changing p Dynamically . . . 94
7.5 Node Failures . . . 96
7.6 Load Balancing . . . 97
7.7 Large Scale Deployment . . . 98
7.8 Frontend Scheduling Performance . . . 101
7.9 Query Delay Comparison: ROAR vs. PTN . . . 102
7.10 Evaluation Summary . . . 104

8 Related Work 105
8.1 Content Based vs. Content Insensitive Distributed Search . . . 105
8.2 Distributed Rendezvous Solutions . . . 106
8.3 Structured Overlays and Peer to Peer Search . . . 107
8.4 Content-Based Publish/Subscribe Systems . . . 109
8.5 Relational Databases . . . 110

9 Conclusions 112
9.1 Contributions . . . 112
9.2 Future Work . . . 114

Bibliography 114

List of Figures

1.1 Basic Distributed Rendezvous 12

3.1 Different Distributed Rendezvous algorithms (n = 12, p = 4 and r = 3) 24

4.1 Basic ROAR store and query mechanisms with $n = 12$, $p = 4$ and $r = 3$. Objects are stored in arches of length 1
4.2 Duplicate matches are possible when $p_q > p$ is used. In this case, $r = 4$, $p = 3$ and $p_q = 4$. 31
4.3 Avoiding duplicate matching in ROAR. 31
4.4 A node failure can cause a query to miss a match. ROAR prevents this by splitting the failed node's sub-query in
4.5 ROAR Scheduling Algorithm: Simple Example 39
4.6 Range Adjustment for Query Scheduling 40

5.1 Bandwidth Consumption Comparison between Index-Based solution and PPS 49
5.2 Data Structures Used by PPS 63
5.3 Running a Query with PPS: System Architecture 64
5.4 Execution traces for queries searching 1 million metadata 67
5.5 Query delays with in-memory data and different number of matching threads 68
5.6 PPS performance scaling with file collection size on a Dell 1950 69
5.7 PPS performance scaling with file collection size on a Sun X4100 70

6.1 Basic Delay Comparison for SW, PTN and ROAR 76
6.2 Variation of Query Delay with N 78
6.3 Variation of Query Delay with Load 79
6.4 Variation of Query Delay with Server Heterogeneity 81
6.5 Algorithm Performance with Different Server Speed Estimation Errors 82
6.6 Increasing p_Q and its effects on the algorithms 82
6.7 Effects of ROAR Mechanisms on Performance 83
6.8 Algorithm Unavailability Comparison for Strict Operations 85

7.1 Effect of p on system performance with PPS_LM 91
7.2 Effect of p on system performance with PPS_LC 91
7.3 Average CPU load for each node 92
7.4 Effect of updates on server throughput 94
7.5 ROAR Changing p Dynamically 95

7.6 Effects of 20 Node Failures on ROAR 96
7.7 Fast Load Balancing with $p_q > p$ 97
7.8 Delay Distribution with Fast Load Balancing when using $p_q > p$ 97
7.9 Range Load Balancing 98
7.10 Effects of Range Load Balancing 99
7.11 Delay Breakdown as seen at Frontend Server 100
7.12 Frontend Scheduling Delay for PTN and ROAR 101
7.13 Observed Server Processing Speeds 102
7.14 Query Delay Comparison ROAR vs. PTN 103

List of Tables

6.1 Simulation Parameters 77
6.2 Bandwidth consumption comparison (messages per operation) 86

7.1 Server Models Used in Experimental Evaluation 90
7.2 Energy Savings running at $p = 5$ instead of $p = 47$ 93
7.3 ROAR performance running on 1000 servers in EC2 100

Chapter 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 [BDH03]. 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*.

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 - such as Google, Bing or Yahoo Search
- Product search provided by online shops such as EBay, Amazon.
- Image search and other complex searches that are difficult to index properly
- The "map" operation of map/reduce computation can be thought of as an instance of distributed rendezvous, where the query is the mapping function to be executed and the data is the input to this function.
- Parallel databases

Successful web search engines such as Google or Bing use parallel index-search algorithms [BDH03], which are a form of distributed rendezvous. The datasets involved can be many terabytes in size [BDH03], 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.

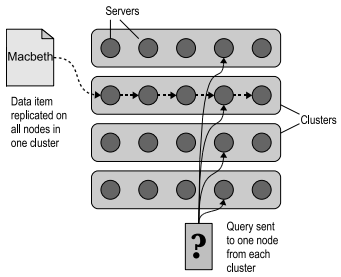


Figure 1.1: Basic Distributed Rendezvous

Given this 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. The good thing with involving more nodes in each search is that it typically reduces the search completion time[1]. On the down side, per query overheads increase with the number of nodes participating in each search: the bandwidth to transmit the query increases, and each of the queried nodes starts a search thread, sends and receives data, etc. In general, these per-node overheads do not depend on the amount of data being searched, i.e. are fixed. Sending the same query to more nodes means that the system is "paying" more fixed overhead per query. This reduces the amount of useful work the system can do, thus reducing throughput.

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. In such scenarios, the system becomes inefficient but keeps running. A worse case is when the index grows so much that the portion of it each server needs to store outgrows the memory of the machine. As Google's web search algorithm runs from memory, it becomes *necessary* to repartition.

[1] assuming delay variance across nodes does not increase significantly

We have focused the discussion on Google to be more specific; the need to repartition dynamically equally applies to other distributed rendezvous applications. In general, it seems that there are two forces driving the need to repartition in distributed rendezvous applications: a) ensuring the system runs as efficiently as possible given the current load, and b) ensuring the system does not hit scaling walls; for instance, this could equate to ensuring it does not run out of memory or disk on any of the servers.

Current systems, such as Google, only repartition infrequently and in response to scaling concerns [Dea]. The repartitioning process is equivalent to restarting the system in the new configuration (although it can be done incrementally; a more detailed description of Google's approach is provided in Chapter 2). None of the deployed systems we know about can (let alone do) repartition dynamically to increase efficiency. We believe this is a result of using cluster-based distributed rendezvous algorithms, that make reconfiguration expensive and difficult. We take the view that if repartitioning were cheap and non-disruptive to the running system, the system would leverage it to increase its overall efficiency.

This thesis sets out to build distributed rendezvous systems that use repartitioning as a knob to control the properties of the system. Specifically, we propose a novel distributed rendezvous algorithm called Rendezvous On A Ring (ROAR) that achieves most of the desirable properties of the cluster-based algorithm while allowing reconfiguration with minimal bandwidth cost.

The main difference between ROAR and the cluster-based algorithm is the way replicas of data are laid out on servers: instead of using clusters, ROAR arranges servers in a virtual ring and stores each replica on a portion of the ring. The biggest gain is that repartitioning now only equates to expanding or contracting these areas. The biggest challenge is effectively using heterogeneous servers to reduce query delay and its variation. The core of the problem is the "sliding window" positioning of replicas ROAR uses. This technique effectively reduces the number of server combinations a query can be sent to, and hence it is more difficult to include faster servers in more queries.

We present the design of ROAR and throughly explore its properties. We present and evaluate several techniques that together overcome the delay challenge. We evaluate ROAR against the cluster-based solution, and against the theoretical best algorithm. Experiments on a 50-server deployment in the Hen testbed, together with experiments on 1000 servers on Amazon's EC2 show ROAR's practicality.

To test ROAR we chose the Privacy Preserving Search (PPS) application. PPS is an application where untrusted servers can match encrypted queries against encrypted data without knowing the contents of the queries or the data. PPS could be used, for instance, to enable privacy in online services such Microsoft Office Live and Amazon S2. This thesis contributes a security model for PPS and novel techniques to support matching numeric predicates against numeric data.

PPS is both disk and CPU intensive and achieving reasonable query delays for large datasets requires parallelization. Distributed Rendezvous in general, and ROAR in particular, is a natural solution to scale PPS to large datasets and high query rates. We show that PPS can scale to millions of items and high query rates by running on more servers.

This thesis is structured as follows. Chapter 2 provides an overview of the problem space, including relevant applications, scope and requirements. Chapter 3 analyses the solution space focusing on the

way different algorithms place their data on servers, and reviews in depth the most relevant existing literature on distributed rendezvous. Chapter 4 presents our solution, ROAR. Chapter 5 presents the PPS application, together with our solution for performing privacy preserving search. We perform an analytical evaluation of ROAR comparing it to other approaches in Chapter 6. Chapter 7 contains an in-depth evaluation of ROAR and PPS running on top ROAR. Related work is reviewed in Chapter 8. We conclude in Chapter 9.

Chapter 2

Problem Space

The Web has expanded enormously since its inception, merely 20 years ago. In this short period, more and more data has become available online, totalling more than 50 billion pages today [web09]. For this huge amount of data to be useful, users need ways to discover relevant information quickly. Search has emerged as a backbone for the Web, supporting the Web's growth by offering a simple interface that allows users to find interesting data.

There are many flavours of search available on the Internet, each tailored to meet different needs of the user. Web-search engines (such as Google, Bing, Yahoo Search) are the most general example: they create and store a reasonably accurate snapshot of the Web, and use it to answer user queries. Online retailers such as eBay or Amazon offer product search to their users, allowing them to find interesting items in large product databases. News sites typically use categories to allow users quick access to desired content, but also offer search to allow for more specific queries. Online data repositories such as Flickr or Picassa (storing photos), Google Docs or Microsoft Office Live (storing documents) also offer search to allow their users quick access to information. In fact, almost all major websites today use some form of search to help users sift through large amounts of data.

Searching large databases is not technically easy. As the datasets involved can be huge, they cannot fit the memory or even disk of any single server. Even if they did, running a query against the entire database takes a long amount of time, much more than the time users are willing to wait for a response. Search systems must provide correct answers fast, typically well under one second. Finally, search systems must be built with scalability in mind: the dataset sizes are constantly increasing, and search volumes will likely increase too. Solving for the current dataset sizes and query loads is not enough. A good solution must be able to smoothly adapt to changing workloads.

To address these issues, most of the existing search solutions rely on two basic tools: **partitioning** the query and **replication** of the data.

Partitioning allows running the query in parallel on many servers. The dataset is partitioned among many servers, such that each server will store a subset of the total data, and all servers collectively store all the data. To run a query, the system will send it to *all* the servers. Each server locally runs the query against its part of the dataset, and returns (partial) results. These results are merged into the final answer, which is sent to the user. For simplicity of presentation, we will refer to sending the query to the servers

holding different parts of the data as *partitioning* the query. However, we note that the query itself is not split in any way, but rather the data that is matched by the query.

Partitioning the query to more servers typically reduces the query search time: the time a server takes to run a query against its data is smaller as the dataset gets smaller. Partitioning more would always reduce delay if end-to-end delay were only determined by local query search times. However, there are other delays that affect end-to-end query delay: network delays in sending the query and receiving the results typically increase with the number of servers. If we partition too much, the network delays will tend to dominate the local query delays, and at this point partitioning further only increases delay.

At first glance, partitioning is work conserving: the total amount of work done is constant, regardless of the partitioning level. This is because there is no duplicated work: each server only works on its unique subset of the data, and no other server works on the same data. Globally, the dataset is only matched once against the query, regardless of the number of servers involved in the search.

However, this view is not accurate: there are overheads associated with starting a query on a server, and these overheads scale up when the partitioning level increases. For each query, each server processes the query message, starts a search thread, waits for the thread to finish, and sends a reply message. This overhead is constant, as it does not depend on the size of the data being searched. Further, the network will send proportionally more messages as more servers are involved in the search. The front-end server—receiving the query from the user and sending it to the query servers—will work harder to schedule a query on more servers (as the complexity is at least linear, as we will show in Chapter 3).

To summarise, the total per-query overheads increase when the partitioning level increases. This is work the system does, but is not useful per se; hence, it negatively affects the maximum throughput of the system. Further, for the same amount of useful work, the system has to do more total work as the partitioning level increases. This increases energy usage at the least; in the evaluation we show this effect for the PPS application.

Partitioning is useful to split the data into chunks that fit on, and can be quickly searched by, individual servers. Partitioning alone is not enough: if any single server fails, the subset of the data it stored becomes unavailable to queries. In such cases, the search system will either return incomplete results or just stop responding to queries. Even if the servers were perfectly reliable, scaling the system with partitioning alone is insufficient. If query load increases, servers are typically added to the system to cope with the additional load. This would cause the partitioning level to increase, which in turn can have negative effects on end-to-end delay and will increase overheads,

Replication helps with both these problems. The dataset is partitioned as before, but the number of partitions is now less than the number of servers. Then, each of these parts will be replicated on a few servers. To run a query , the system will send it to enough servers that hold all the data between them.

Having more than one replica of each data part increases fault tolerance and availability. Adding servers to the system, and loading these servers with replicas of existing data is the easiest way to scale the system up when query load increases.

Replication comes with its own overheads: the more replicas of the dataset, the bigger the update

cost (in terms of network traffic and local processing). Typical search databases do not change that often, so this may not be a big issue. However, there are datasets that change frequently (e.g. news); in such cases, the update costs may become a limiting factor in system performance.

As with partitioning, replication alone is not sufficient to support search. When the data set grows, the amount each server has to store and process grows. Veritable scaling walls will be hit when the amount of stored data exceeds any individual server's capacity; also query delay will increase to unacceptable levels when the data set becomes too large. Finally, having more replicas to update will increase the time needed to reach consistency in the system.

The main problem with existing search engines is that they make it difficult to change the replication or partitioning level. The typical mode of operation is to estimate dataset size and update frequency, as well as query load, and use these to statically compute the required levels of partitioning and replication. If these parameters require changing, the system is effectively "restarted" with the new parameters [Dea].

This by itself is not a big issue if these estimates are accurate, but they rarely are: query loads, in particular, are notoriously bursty (e.g., due to "flash crowds"), the dataset sizes continuously increase and their update rate changes. The task then is to "optimise" the parameters a-priori. If one uses worst-case predictions to derive the parameters, the system is prepared for the worst but is highly inefficient. If one uses average case predictions, the system will fail to meet its targets (such as delay) when load exceeds expectation, and will be inefficient when load undershoots. Hence, it is important to allow system **reconfiguration** at runtime.

Current systems repartition infrequently [Dea], typically when the dataset size exceeds the memory of the nodes. Replicas can be added as query load dictates. These reconfigurations typically require manual input and disrupt the system. This means that such events are the exception, rather than the norm.

Contribution. We take the view that fine-grained adaptation of the replication and partitioning levels can increase the overall efficiency of the system, and will allow systems to cope with wider ranges of traffic loads while maintaining good service.

For reconfiguration to become *adaptation*, it must be fast, automatic and seamless. The system must be able to service running queries with minimal disruption while reconfiguration is taking place. Current algorithms, in particular the Google one, fail to achieve these requirements.

In this thesis we examine the fundamental algorithmic reasons that prevent the existing solutions from achieving adaptation and design a novel algorithm that meets these goals.

2.1 Problem Definition

Distributed Rendezvous (DR) is a solution to search problems that uses the two basic tools of replication and partitioning. Essentially, distributed rendezvous aims to meet (rendezvous) each query with all the data, in a distributed way. It is an algorithm that decides how data is split and how queries are routed to meet the data. It does not dictate how the query is performed locally, on each server (which depends on the type of search provided).

We use the term "distributed rendezvous" instead of "distributed search" for two main reasons. Firstly, we want to focus on the distributed algorithm storing and splitting the data and routing the queries to servers, not on the search algorithm used locally to select the results. Secondly, there are other applications besides search that can use distributed rendezvous: online filtering of content is one such example [RRH07], and the map operation in map-reduce computation is another.

Distributed rendezvous is "dumb" because it does not use content to decide where it should store certain data, nor does it use query content to route the query to the interesting data. It routes the query to meet *all* the data, which is, in a sense, a distributed version of brute force matching. Locally, however, each server will typically use smart algorithms to get the query results from its local dataset.

How come distributed rendezvous-like solutions are used in practice by Google, Microsoft, etc.? Surely, smarter content-based solutions are preferable! The biggest advantages of distributed rendezvous over content-sensitive solutions are its simplicity and generality. We provide an overview of alternative solutions and outline the reasons for DR's widespread use in Section 8.2.

Definition 1 (Distributed Rendezvous)**.** Distributed Rendezvous is a class of distributed algorithm that takes a collection of n servers and a parameter r, the replication level for data objects. It offers two basic operations:

1. **Store Object**: takes as input a data object and stores it on r servers.
2. **Run Query**: when presented with a query, it will forward the query to enough servers to ensure all the data objects are queried.

A third operation that may be implemented by distributed rendezvous algorithms is the ability to change r on the fly. On request, replicas will be added or deleted to achieve the desired replication level; meanwhile queries should still be serviced as usual, possibly with a reduction in capacity.

To achieve full query functionality, a few other operations must be implemented on top of distributed rendezvous: each node must locally search its data objects with the given query, and find matches; results must be sent back to the user. We intentionally leave these out of the DR definition, as they are application specific.

Brewer [Bre01] defines *harvest* and *yield* for distributed rendezvous systems. *harvest* is the fraction of data objects a query visits. When harvest is 100%, all data objects are visited so the query is given an exact answer; when harvest is less, an approximate answer is returned. *yield* is the number of queries that are serviced out of the total number of queries. Ideally, we would like to service all queries and thus have $yield$ close to 100%. However, when systems are overloaded it may be desirable to drop some queries altogether to ensure the rest of the queries are executed.

Comments on Def. 1. By definition, we require harvest to be 100% as some applications will require exact answers.

We intentionally choose the same replication level for all objects. This is the next step after not considering content: we treat all data objects equally. Some applications may need to give more weight to certain objects if harvest is less than 100%; this is to ensure that some important objects are always

visited. These mechanisms can be layered on top of a distributed rendezvous algorithm as needed, without changing the underlying behaviour for the majority of objects. Hence we consider them orthogonal to the basic distributed rendezvous problem.

In practice, when designing a distributed system, one asks the question: how many servers are needed to support a certain query throughput, given a collection of data objects? Here, we take the dual approach where we assume n, the number of servers, is given and and ask what is the maximum query throughput. An answer to the latter question implies the answer to the former. We prefer the dual formulation as it allows us to reason about the properties of the algorithms more easily.

A secondary question is: what is the proper replication level? The common approach is to choose the replication level a-priori according to requirements such as availability. We take the view that the optimal replication level is difficult to characterise beforehand and may even change as the system evolves. Therefore we choose to expose r as a knob to the application.

Definition 2 (Partitioning Level)**.** Given n and r, let the partitioning level p be the minimum number of servers a query must visit such that it collectively meets all the data objects.

Definition 3 (Load Imbalance)**.** Given an assignment of items to servers, let $assigned_i$ be the number of items assigned to server i. We define load imbalance lb as:

$$lb = \frac{\max_{i=1}^{n} assigned_i}{(\sum_{i=1}^{n} assigned_i)/n}$$

In short, load imbalance is the ratio between the maximum load assigned to a server and the average load. When items are split eventually among servers the imbalance is 1; when all items are assigned to a single server the imbalance is n.

In the definition above, items can be either replicas or queries.

2.1.1 Running a Query

Splitting work among multiple servers serves to pool the resources of those servers together, making them act as a single resource. DR can help pool the disks, the CPUs and the memory of the nodes. In this section we aim to create models of how these resources can be shared.

Definition 4 (Object)**.** An object is a collection of bytes that is stored in the DR system. It has an identifier associated with it, uniformly distributed from an object identifier space. For example, assume this space is unsigned integers.

Definition 5 (Query)**.** A query is a predicate (a polynomial time computation) that takes as input a data object and answers yes/no indicating whether the object matches the predicate or not.

To execute a query is to run it against all stored objects with IDs in the object identifier space. To partially execute a query is to run it against objects with IDs in a subset of the identifier space. To split a query on many servers is to partition the identifier space into (a few) sub-queries, and assign them to different servers.

Definition 6 (Splitting a query)**.** The query in Distributed Rendezvous contains partitioning information that informs servers which of their objects should be matched against the query.

The query does not need to specify exact object identifiers, but rather the range of objects this query needs to match.

2.2 The Distributed Rendezvous Trade-off

What is the relationship between r, p, and n? The answer depends on load balancing.

Let us consider the case when we have perfect load balancing for object replicas (i.e. $lb_{data} = 1$). Let the number of data objects be D. Each server will store approximately Dr/n objects. To achieve correctness, a query must visit enough servers such that it visits all data objects. Dividing D by the number of objects on each server, we find that the minimum number of servers to be visited—the partitioning level p—is $p = n/r$. Thus:

$$rp = n \tag{2.1}$$

Another way to think of this equation is to place the n servers in a matrix with r rows, where each data object is stored in all the servers from one random column. When a query arrives, it must visit all the servers in a random row from the matrix.

This is the main trade-off distributed rendezvous offers. There is a direct relationship between the number of replicas for each data object and the number of servers a query must visit. As a consequence, when parameterizing DR we can choose either r or p; given n, the other is implicit.

Increasing r increases availability, and also increases the ability to avoid slow servers - thus improving query delay. Smaller r means less bandwidth is used for storage, as each object update needs to be sent to fewer servers.

We note that p is the minimum number of servers that ensures correct query execution. In reality, values larger than p can be used for any given query. If utilisation is low, using higher values of p will result in lower delay for CPU bound queries, as long as duplicate matches are avoided across servers.

In distributed rendezvous r is no longer just a tool for increasing availability, but a way to control other properties of the system. For instance, to achieve consistency, chain replication [vRS04] serialises updates at the first replica and only allows reads at the server hosting the first replica. DR only settles for eventual consistency but can instead use r to affect other properties of the distributed system such as bandwidth consumption or delay.

2.3 Scope

What are the DR-like problems we are trying to support? Answering this question will help us narrow the large problem space.

We envisage two main categories of problems:

- **Query Applications.** Here, a query is presented to the distributed system, which executes it and returns the results. Examples are many: Google/Microsoft/Yahoo web search, eBay/Amazon product search, Privacy Preserving Search, etc. We assume this service will run in a data-center, with low delay and high bandwidth between servers.

- **Online Filtering Applications.** Here, users express their interests which are stored in the database. When new documents arrive, they are matched against existing interests and forwarded to interested users. This is the dual of the query scenario: user interests are stored instead of documents. Such a system could be used to quickly disseminate RSS feeds. A possible deployment is the Web servers themselves.

Different applications have different bottlenecks. The privacy preserving search application we experimented with is CPU bound. Web-search seems to be CPU or memory bound [Dea]. other types of searches are disk-bound. The online filtering example may be bandwidth bound, as documents must be sent to many users.

The main focus of this thesis is the first type of application.

Although there are many potential constraints on creating a solution, three are of particular importance:

- the reliability of the servers, which affects the availability of the system;
- the cost of communications between the servers; and
- the acceptable delay bounds of the application.

For any particular stable scenario, with a particular ratio of stored objects to queries, it is possible to choose a value of r which optimises these objectives. For example, Google might wish to minimise bandwidth costs, subject to the threshold constraints of one second search latency and five-nines availability. However, if conditions change, given that r is the only free parameter, it may be important to change the replication level *on-the-fly* to re-optimise the system.

2.3.1 Server Reliability

The applications we envisage, while distributed, are most likely to run on maintained servers with relatively stable overlay membership (i.e. servers are allocated to the search application for a long time) and infrequent failures. Thus we prefer solutions suited for comparatively low churn and failures.

Different algorithms hit different points in the availability/cost/delay tradeoff space. However, regardless of the algorithm, increasing r increases availability. Applications can monitor availability when running queries and adapt r appropriately.

Failure Model. We assume server behaviour is fail-stop, and that failures are independent.

Data Consistency. We assume lazy replication is used to store objects, and thus eventual consistency is achieved. This is appropriate for the target applications, where objects updates are comparatively rare to query rates, and a few transient false positives or negatives are acceptable.

2.3.2 Communication Costs

Bandwidth is often the bottleneck in wide-area deployments and even between racks in data centers [DG04], and thus it may limit the query-processing rate. In addition, many applications care about throughput, a prime example being map/reduce systems, where queries are executed offline against stored data. The online filtering application is bandwidth-bound if run in a wide-area setting.

In general, if bandwidth costs are high then reducing bandwidth usage is an objective. While this is obvious in the wide-area, this may be even true even in data centers with scarce inter-rack capacity available, and the new focus on energy consumption: reducing bandwidth usage also reduces energy related costs [NPI+08].

How does bandwidth usage depend on r? Let B_{data} be the bandwidth of incoming data object updates and inserts. Let B_{query} be the bandwidth of incoming query traffic, and $B_{results}$ the bandwidth used by query replies. The total bandwidth consumption B of the system is: $rB_{data}+pB_{query}+B_{results}$. $B_{results}$ does not depend on r, p or n and can be considered constant for this analysis.

Using Eq. 2.1, it is straightforward to show that the value of r that minimises the bandwidth is $r_{opt} = \sqrt{n \cdot B_{query}/B_{data}}$.

If we sub-optimally chose an extreme value of r, either very small or very large, this requires $O(\sqrt{n})$ more bandwidth than optimal.

2.3.3 Application Delay Bounds

While interactive applications care about bandwidth costs, their primary concern is keeping query times short. When DR partitions a query, latency is reduced as p servers work in parallel on disjoint subsets of the data. The larger p is, the smaller the resulting delay.

Further, depending on the overall load, running the same query on the same number of servers may take a different amount of time. For instance, approximating the system with an $M/D/1$ queue, waiting time increases with load (ρ) as $\rho/(1-\rho)$.

We can effectively write a function $minP$ that takes as input the servers' processing capacity and the load in the system, and outputs the minimal value of p that achieves the target delay. For different values of load, $minP$ will be different.

Chapter 3

Solution Space

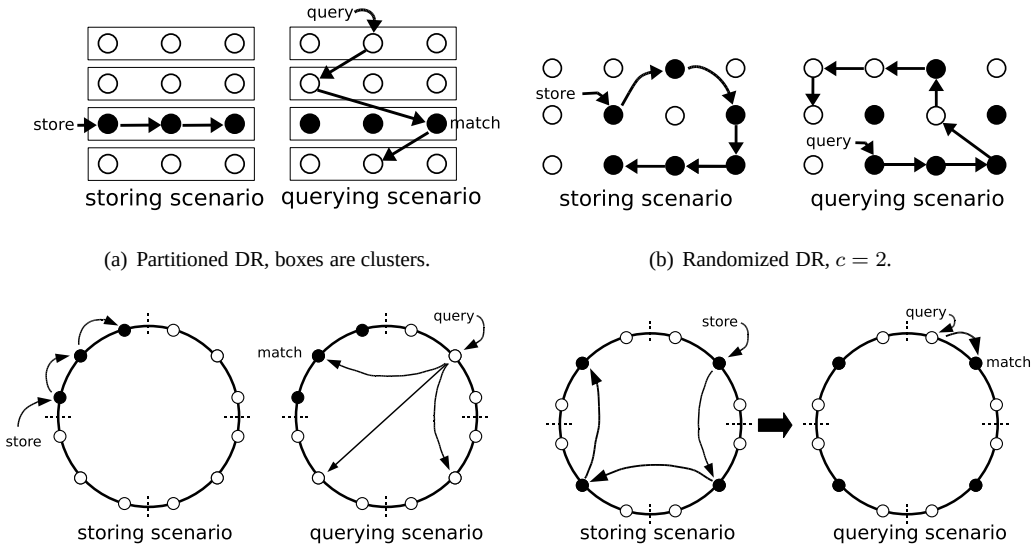
Our end goal is an algorithm that can easily adapt to changes in both query rates and data update frequency, providing bounded search times as efficiently as possible. Efficiency favours using partitioning and replication levels such that $pr = n$. Within this envelope, lower p values are preferred if query rates are high, and lower r values are preferred if data updates are frequent.

Our first requirement for a candidate solution is to efficiently support the basic functionality : running queries and storing data ($pr = n$). The algorithm must allow changing the ratio between p and r when the number of servers n is fixed, while at the same time allowing us to change p and r by adding or removing servers.

Adding and removing servers is the basic way to scale systems up and down. We are introducing a new load adaptation technique that changes the ratio between p and r, while keeping n fixed. Such reconfiguration is desirable when the new load can be handled by reconfiguring the existing servers, and is the only viable strategy when additional compute capacity is not available. Even when additional compute capacity is available, powering up new servers and bringing them to a consistent state can take minutes. Adapting the ratio of p and r can provide a seamless experience to the users until the new servers are up and running.

What are the possible Distributed Rendezvous algorithms to achieve these goals? Many different solutions are possible, each choosing a different set of design trade-offs. The basic solution must dictate how to split data into parts and where to replicate each part as well as how to run the query such that all the data is visited. These two components are tightly related; choosing a solution for one constrains sensible choices for the other. It is easier to reason about replica placement strategies, so we start our solution exploration here.

In their study of reliability, Yu et al. identified three main classes of replica placement algorithm [YGN06]: *Partition*, *Random*, and *Sliding Window*. We will analyse algorithms from these classes in our quest to obtain a practical DR algorithm. In this chapter we develop skeleton DR solutions for each of these classes, specifying replica placement strategies and ways to dynamically adjust the replication/partitioning level. A valid question arises: are these algorithms diverse enough that they capture all the interesting parts of the solution space? To reassure ourselves that these algorithms cover a wide enough spectrum, we informally discuss optimality criteria and how these algorithms score against them.



(a) Partitioned DR, boxes are clusters.

(b) Randomized DR, $c = 2$.

(c) Sliding Window DR.

(d) Dual Sliding Window DR.

Figure 3.1: Different Distributed Rendezvous algorithms (n = 12, p = 4 and r = 3)

In our analysis (chapter 6) we perform a thorough comparison to the optimal solutions.

We conclude this chapter with a brief high-level comparison between these different DR solutions.

3.1 Partitioned Distributed Rendezvous

Intuitively, the simplest way to organise servers is in a matrix where each replica is stored on one row of servers and each query is executed by a column of servers. The problem with this view is that, in reality, r (or p) rarely divide n and thus we do not have a perfect matrix. The solution is then either to run queries not necessarily in a column, but on one server in each row; or the dual is to run queries on a column, but store replicas once in every column. The first strategy is better for a number of reasons, so we discuss it first.

The Partitioned (PTN) strategy is parameterised by p. It divides the n servers into p clusters each with approximately n/p servers; each object is then stored on all the servers in one randomly chosen cluster. For routing, queries are sent to one server [1] in each cluster (see Figure 3.1(a)). This is the algorithm used by Google [BDH03]; we call it PTN.

Partitioned algorithms can change p, but the change is disruptive. To decrease p, servers must first be freed to store the new replicas, which is done by destroying a cluster. Each object from this cluster must be stored on all servers in one of the remaining clusters. Finally, the free servers can be split between the remaining clusters and each must then retrieve a copy of the objects stored in its new cluster.

Increasing p is simpler: a few servers from each cluster simply leave and form a new cluster. Initially this cluster will store no data, so to improve load balancing some objects can be transferred to it from existing clusters.

[1] Typically this choice is influenced by load balancing.

Coordinating the change of p remains an issue. In a data center, one server can simply be chosen as the master. Changing p in a distributed manner is more difficult because the actions of servers differ: some need to leave and re-join, others must replicate objects, and some others must delete objects. Just designating which servers should leave a cluster requires some form of distributed agreement. The network structure also needs to change when clusters are created or destroyed; depending on how this is done, it may add additional costs or result in some missed queries during the transition.

Finally, r or p can be changed by modifying the number of servers. When increasing n, it is simplest to add the new servers to the existing clusters, thus increasing r. In theory it is possible to create a cluster with new servers (i.e. increase p) and repartition the existing objects, but this only works when a whole new cluster is added; otherwise, a small cluster can adversely affect the performance of the whole system. When n is reduced, either r or p can be easily controlled by removing servers from each cluster or destroying entire clusters.

What type of query delay does PTN experience? We are not ready to answer this question just yet, but we just observe the number of choices the algorithm has when assigning a query. For each of the p query parts, the algorithm can choose between r servers; thus the number of possible combinations is r^p.

To achieve maximum throughput, PTN needs to make sure that clusters are computationally equivalent (so that none becomes a bottleneck until they are all fully utilised). That means the sum of processing speeds of servers in each cluster is roughly constant across all clusters.

A dual PTN approach.

Instead of creating p clusters, and having all machines identical in each cluster, we can create r clusters instead. Each data object will be stored r times, once in each cluster. Inside a cluster, a data object will go to a random machine in the cluster. When a query arrives, it is executed by all the servers in a randomly selected cluster.

The nice thing about the original PTN approach is that all servers in a cluster are identical: they store the same replicas. This seriously simplifies cluster management, and is not the case with the dual approach; the dual approach is suited for multiple data center deployments, where replicas are clustered geographically and it is possible to run a query completely inside a data center.

This special case can also be handled optimally with the original PTN: simply create a higher level partitioning of the servers into data center clusters, and then apply PTN inside each cluster of servers. This is simple enough, and yet it reaps all the benefits of the dual approach.

We have run analyses of this dual approach and found performance similar to PTN in all cases except the multiple data center scenario described above; for simplicity of presentation, we omit these (rather obvious) results. We do not consider the Dual PTN as a candidate solution for DR.

3.2 Randomized Distributed Rendezvous

In the Randomized (RAND) strategy replicas of objects are placed randomly on servers during a random walk through the overlay of length $c \cdot r$, and queries are routed to $c \cdot n/r$ randomly chosen servers, as in [TKLB07, FRA+05] (see Figure 3.1(b)). c is a constant of the algorithm.

Unlike the other algorithms we discuss, harvest is not necessarily 100% (i.e. not all objects may be visited by a query): randomized strategies only give a probabilistic guarantee that a query will return all matches. The constant c serves to tune the probability of a query missing a stored object. The typical value for c is 2, which yields a harvest of 98%.[2]

Randomized algorithms can easily change the replication level. To increase r, the last server of each object's random walk simply replicates the object to an additional server. To decrease r, the last server of the random walk discards the object. For the probability of a miss to remain unchanged, all servers (including the frontend servers) need to agree on r so they can maintain $p = n/r$ for their queries; typically, a gossip protocol is used for this.

When n is increased, either r can be increased (by creating new replicas of existing objects), or p can be increased (by moving some existing replicas from running servers). Similarly, when servers are removed r or p are decreased.

The randomized DR algorithms give probabilistic coverage guarantees and have higher costs than deterministic algorithms such as PTN: each query is sent to twice the number of servers, and each object is also stored on twice as many servers for $c = 2$. For these reasons, and despite the ease of changing the r/p tradeoff, they seem of little practical importance to data center deployments, and will receive no further attention in this thesis. They do offer, however, high robustness in face of massive server failures and thus seem better suited for peer to peer type deployments; BubbleStorm is a randomized algorithm suited for these deployments [TKLB07].

3.3 Sliding Window Distributed Rendezvous

An important 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.

The simplest solution in this case 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 3.1(c). 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). 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 furthest around the ring that currently stores it.

[2] While sufficiently high harvest is achievable with small c, 100% harvest is impossible to guarantee unless the query is flooded to all nodes.

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 comes with shortcomings. First, while it works very well with a fixed number of reliable nodes, it does less well if a node fails. If such a failure happens, queries that would have visited this node will no longer match its data items, so some fast recovery mechanism would be needed to replace the failed node. In the meantime, queries would have to visit both the preceding and subsequent nodes to ensure that all data items continue to be matched, causing load concentration on these nodes. In addition, as the data set changes over time, old data items disappear and are not perfectly replaced by new items. Thus the initially perfect load balancing degrades over time.

More generally, the most basic problem with the 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 (either 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 utilise heterogeneous servers according to their capabilities.

How many choices does SW have when assigning a query? SW can only choose the starting point for each query, as this determines all the other points where the query hits. This means we only have r choices. This is much smaller than PTN's r^p choices, so we expect query delays to be higher for SW.

Let the above be the optimal sliding window algorithm, SW.

A dual SW approach

The SW algorithm stores replicas on r successive servers on the ring. Its dual is to store each data object on r equidistant points on the ring, while running each query on all the servers with ids in an arc of size $1/r$ on the ring. This is the approach used by Glacier [HMD05], and is presented in Fig. 3.1(d).

Changing the replication level in the dual SW algorithm is more complicated than in SW. Because replicas are equidistant, on each replication level change some replicas will change servers, and will need to be relocated; a simple analysis shows that if we have n servers, a $1/n$ fraction of objects will need to be relocated on each change.

Even worse, to implement this relocation, the nodes need to remember which node holds the previous and next replica for each object they store. Checking liveness of adjacent replicas implies probing quite a few nodes.

In contrast, in SW the previous and next replicas are implicit (stored on the predecessor and successor); no objects are relocated when r changes; and monitoring liveness is much easier, as it suffices

to monitor the immediate neighbours.

In a distributed setting, running queries is simpler for the dual approach if queries are run recursively (i.e. forward to next neighbour). In reality, queries will be run in parallel to minimise delay, which means the querying node will have to know and contact a few other nodes directly; hence the complexity is similar in both SW and its dual approach. Given its complexity, we drop dual SW from our candidate list.

3.4 Limitations of Existing Solutions

The three algorithms we presented are suited for different parts of the problem we are attacking. RAND stands out as it is designed for peer-to-peer like deployments, with highly unreliable, high churn populations of servers. Such deployments are completely different from the data center deployments we envision, with low churn and failure rates. To achieve reliability, RAND increases the basic Distributed Rendezvous costs. For instance, if we want each query to visit 98% of the data, RAND will spend four times more resources than optimal. At the end of the day, this means we need much more hardware to cope with the same data and query rates. Thus, it does not make sense to use RAND in data-center deployments.

PTN is simple from an administrative point of view and has low basic costs for data storage and queries. It has good load balancing and efficiently supports changing r or p by adding or removing servers. PTN's main drawback is its approach to changing the ratio of p to r when n is fixed. PTN transfers more data and takes longer on each reconfiguration. PTN also reduces overall capacity while this change is taking place. These limitations arise mainly from the asymmetric workload imposed on servers during reconfiguration: a subset of servers will drop their data and reload new data, while the others do nothing. This behaviour emerges from the cluster structure itself and is fundamental to PTN. It is the price PTN pays for simple administration.

SW too has low basic costs for storage and queries. In contrast with PTN, it naturally allows changing of p and r when n is fixed. The process simply involves extending or reducing the replication range of each object, and transfers the minimal amount of data required for reconfiguration. In contrast to the asymmetry in PTN, each server plays an equal role during the reconfiguration process. SW has other problems, though. It does a poor job of load balancing and copes badly with adding and removing nodes, as well as node failures.

In this thesis we will show it is possible to elegantly solve the issues of PTN and SW with a handful of techniques. The resulting algorithm, ROAR (Rendezvous On A Ring), achieves the best of both PTN and SW: it allows easy reconfiguration as SW does, while providing good load balancing and coping with server churn and failures.

Chapter 4

ROAR: Rendezvous On A Ring

The problems mentioned above led us to develop a new algorithm that we call Rendezvous On A Ring (ROAR). ROAR uses the sliding window arrangement of replicas while avoiding its drawbacks. ROAR's insight is that the discreteness of replica placement is the main source of problems. In basic SW, replicas of an object will be stored on r servers from a given starting point. When servers leave, the replicas need to adjusted accordingly, causing a lot of churn.

Rather than simply arranging servers in a circular list, ROAR uses a continuous circular ID space (for simplicity assume its range to be $[0, 1]$). 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 [SMK+01], although that is where the similarity ends.

To decouple replica placement from server replication, we define for each object a continuous range on the ring called "replication range". The object will be stored on all servers whose range intersects its replication range. When a server leaves no operations are necessary to ensure consistency, as objects' replication ranges are constant; some objects will simply lose one replica. Similarly, when a node joins, it will load the objects it should store; replicas stored on other servers will not be affected.

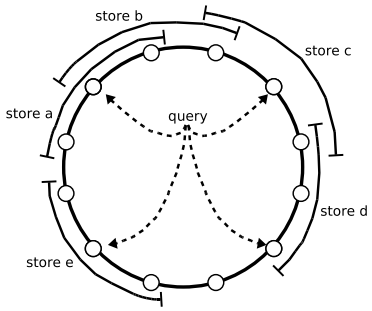


Figure 4.1: Basic ROAR store and query mechanisms with $n = 12$, $p = 4$ and $r = 3$. Objects are stored in arches of length $1/p$ and queries sent to p servers at $1/p$ intervals, thus ensuring that a query visits all stored objects (denoted by letters).

The partitioning level p defines the length of the replication range. Given p, ROAR stores each object on the servers whose range intersects an arc of size $1/p$ on the ring (the replication range, see figure 4.1); 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.

While the basic concept is very simple, there are a number of details that matter for correctness.

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. Several strategies are viable, depending on the deployment scenario:

- 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 [GGL03]) where the filenames embed the node identifiers. 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 backend update server that knows the ring topology.

A peer-to-peer solution using ROAR might use the first, whereas organisations with existing distributed filesystems might choose the second. Our implementation uses the second, with NFS as a filesystem and a special file structure to store the objects (see Section 5.6 for details).

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.

We first discuss the simpler case when all servers are equally powerful; the general case is discussed in Section 4.8.1. 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 4.1). 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

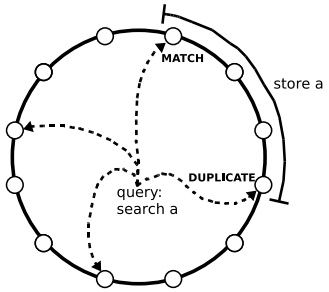


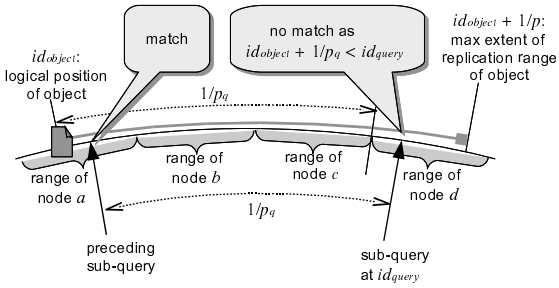
Figure 4.2: Duplicate matches are possible when $p_q > p$ is used. In this case, $r = 4$, $p = 3$ and $p_q = 4$.

in figure 4.2). 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:

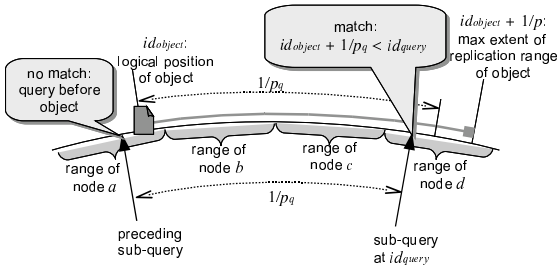
$$id_{object} < id_{query} \tag{4.1}$$

$$id_{object} + 1/p_q >= id_{query} \tag{4.2}$$

Data items that do not satisfy the second condition will be matched by the preceding server that received a sub-query (figure 4.3(a)), while data items failing the first condition will be matched by the server receiving the following sub-query (figure 4.3(b)).



(a) Match by first sub-query



(b) Match by second sub-query

Figure 4.3: Avoiding duplicate matching in ROAR.

Why then is it so useful to be able to run queries with values of p greater than the bare minimum needed to match all data items? There are two main reasons:

- 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 basic partitioning to be changed while still serving queries.

4.3 Adding Nodes

To be able to function correctly, each server just needs to know its ID range, and this should match up with the ranges of its immediate neighbours on the ring.[1]

When a server joins the system, 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 neighbour. We discuss other insertion strategies, as well as a practical way to implement them in Section 4.9.

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

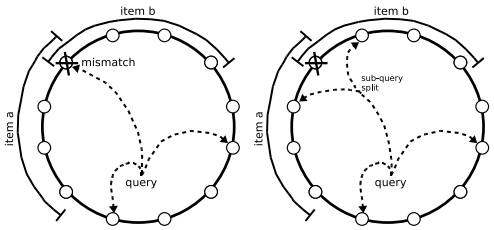
Once the data download has finished, the new node communicates directly with its two neighbours to determine which of them is most loaded. It now starts to grow its range into that of the most loaded neighbour, requesting additional data items that overlap the range as it grows. Every few seconds it updates the front end servers with its new range, and also updates its neighbour so that the neighbour 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 neighbours, the rate of replication is slowed to a low background rate. In fact, nodes *always* compare load with their neighbours and expand their range very slowly into that of a more loaded neighbour. 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 neighbours that its load is now infinite. The two neighbours 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: if each data item is replicated on r servers on average, then $1/r^{th}$ of the data on a node starts or finishes at that node; it is this data that the neighbour will not already have. If a node has k data items already and its neighbour wants to shut down, it will need to download $k/2r$ new data items if it takes

[1] This is not always strictly required for correctness, but is needed for efficiency.



(a) A failure causing a mismatch.

(b) ROAR's failure-handling mechanism.

Figure 4.4: 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.

over half of the neighbour's range.

The query load will increase by as much as 50% on the neighbours of the node being shut down, as their range has increased by 50%. However, in practice the neighbours' neighbours 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. We could simply choose initial values for the start of the query on the ring so that the failed node is not hit, but this would reduce the overall capacity by a fraction of $1/r$ for a single failure, and might be infeasible for multiple failures.

Instead the front end server ignores the failures when deciding the starting point of the query, but 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 on the failed node. 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.

The general idea is shown in figure 4.4. The first subfigure shows how a failed node causes queries to miss a match against items a and b. The second subfigure depicts ROAR's fall-back strategy, whereby a sub-query meant for the failed node is split in two and sent to the failed node's predecessor and successor nodes. The former is needed in case the item's range ended at the failed node (as is the case with item a) and the latter in case the item's range started at the failed node (item b). To maximise the load spreading, 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. Choose a new first sub-query destination 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 destination id_{q2} such that:
   $id_{q2} = id_{q1} + (1/p - \delta)$
   This guarantees the new sub-queries are not so far apart that a data item can fall between them. Thus all data items will be matched.

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, 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 general 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 $p_q \cdot r >= n$. 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 incurs extra bandwidth costs, and eats CPU and network bandwidth 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 catch up 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. This may be limiting throughput, but in any event it is costing CPU cycles and hence increasing energy requirements[2].

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,

[2]The reader may think that the effect is negligible, but the temperature in our air-conditioned machine room runs 4° Celsius hotter when our 43 ROAR nodes are fully loaded than when they are idling.

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: Proportional Ranges

ROAR performs load balancing by adjusting the size of the segment of the ring that a node is responsible for. With queries, the mean query rate seen by a node is directly proportional to the node's range g_i. As mentioned previously, ROAR evens out load by a slow background process in which each node extends its range into that of a more loaded neighbour. 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.

With stored data items, if the ROAR system indexes D items in total, the number that need to be stored on a node with a range of size g 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 $D/p + D \cdot g_i$. On average $1/p = r\bar{g}$, so for sensible values of r, the D/p term dominates, and the amount of data stored by each node is fairly even between nodes.

Our discussion in Section 4.2 assumed all servers are equally powerful; in that case randomly choosing the starting point[3] gives perfect load balancing and minimal average delay.

When servers are not equally powerful, assigning larger ranges to faster servers allows perfect query load balancing, whereby each server will serve queries according to its processing capacity. In this way, servers are uniformly loaded and no server is a bottleneck until the system as a whole cannot support the query load. However, when the system is lightly loaded, perfect load balancing is not needed. In such cases, it is best if we run the p sub-queries on the most powerful p nodes, as this minimizes query delay.

PTN naturally optimises for both load balancing and query delay at the same time: load balancing is ensured by having p equally powerful clusters of nodes, while query delay is minimised by choosing in each cluster the server that would first finish the sub-query. ROAR load balancing is given by unequal node ranges. To minimize query delay, the ROAR scheduler considers different starting points for the query and picks the starting point that finishes first (we present an algorithm that achieves this in section 4.8.1). Compared to PTN, ROAR has a lot fewer choices in its selection of servers to run the query: it must choose between r configurations. In comparison, PTN has to choose between r^p configurations; that is why PTN has better delay than ROAR. In the next section we show how to change the basic ROAR to get better query delay.

Another way to get better query delays and to improve query load balancing is to increase p_q, the number of servers that run each query. This is not the preferred way, as it increases overheads due to sending queries. However, it can be selectively used to ensure that certain max query delay bounds are met; we present a heuristic algorithm for this purpose in section 4.8.2.

[3] Or choosing the lightest loaded of two to smooth out load, as in the power of two choices [Mit01]

4.7 Multiple Sliding Windows

To improve query delays for ROAR, we use a simple variation that makes it more PTN-like. Instead of having all servers belong to a single logical ring, create a small number of rings (say 2) and have each server belong to only one of the rings. Objects would be stored in both rings, with $r/2$ replicas in each. A query would still touch p equidistant points, where each point belongs to either of the rings.

Because p is the same, on average each object still has r replicas; adding a second ring does not add overhead for storing objects or running queries. It does, however, mandate that any object has at least two replicas (as it is stored once on each ring), so r cannot be lower then 2.

For availability purposes, $r \geq 2$ anyways, so this is not a drawback. If we used more rings, this limitation would become important. At the extremes, we could create r rings. This turns the ROAR algorithm into the Dual PTN algorithm, so we lose all the benefits of SW to easily change the tradeoff.

With two rings, ROAR has more choices when running queries, namely $r \cdot 2^{p-1}$. This is much better than SW's r choices, but less than PTN's r^p. We show in simulation that using multiple rings increases availability when search operations are strict, i.e. all objects must be visited by a single query for the query to succeed. Also, multiple rings allow much simpler adaptation to daily load fluctuations, as we discuss in Section 4.9.1.

4.8 Running Queries on Heterogeneous Servers

As mentioned previously, the front-end server receives the query from the user, splits it and runs it on the ROAR nodes, and finally aggregates and returns the result to the end-user. The front-end logs query delays, and controls p - the query partitioning level. The front-end server also maintains statistics about each ROAR node:

- The node's range (which implies the node's minimum value of p)
- Node's liveness (last time seen up)
- The outstanding queries scheduled on the node, and their expected finish time
- The processing speed of the node (this includes other background load not from ROAR)

When a new query arrives, the front-end will split it into p sub-queries. Using information about outstanding queries and node processing speed, it decides which servers should process the query and sends the query to those servers, setting timers for each sub-query. These timers are used to detect node failures quickly: if a query response times out, the node is marked as dead. The unfinished sub-query is split further into two smaller sub-queries that are rescheduled on the failed node's neighbours.

As results return, the front-end assembles the reply; when all results are received, the query is marked as finished and its finish time recorded; the results are sent back to the user. Also, estimates are made for each sub-query for the processing speed of the server, and an exponentially weighted average processing speed is updated with the new data.

ROAR needs to send queries quickly and reliably to the matching servers; our implementation uses TCP for reasons explained below, but other choices are possible.

4.8.1 Scheduling Algorithm

ROAR uses server processing speed estimates together with sub-query size to estimate sub-query execution times. ROAR does not model network delays, as in data centers round trip times are well under 1ms, being negligible compared to query execution times. It is straightforward to extend this algorithm to take into account network-induced delays.

We describe the scheduling algorithm for the single ring version of ROAR first, and show how it can be extended to support multiple rings later. ROAR has to choose a starting point for each query to minimise the delay. There are r possible combinations of servers that can be chosen. To test all combinations, it suffices to pick the starting point of a p way query in the first $1/p$ range of the ring; this is because all the other $p-1$ are equidistant, sweeping different parts of the ring. The simplest algorithm is to choose one or a few random starting points and use the one that gives the smallest delay. To get perfect results, however, we need to pick many random starting points (a lot more than r).

A deterministic approach is the following. Let id be the starting point of the query. Iterate with id from 0 to $1/p$ increasing id with a small $delta$, computing the expected query delay, and choosing the fastest point. To get all possible combinations we need a small $delta$; however, on each iteration we need to find out which node is in charge of each of the p points, and compute the delta; this significantly increases costs.

Our final algorithm dynamically changes $delta$ to minimise the number of iterations. The insight is to move id at each step by enough to hit at least one different server in the "selected" configuration. The finish times for all the other servers are already known; all we need is to compute the finish time of the new server, and check if it affects the overall query delay. The pseudocode is given in Algorithm 1.

The algorithm has an initialisation phase where sub-query and total query delays are computed for $id = 0$. The function *node_in_charge* does a binary search through the list of server identifiers to find the server in charge of a sub-query. The function *estimate_finish* uses server speed and load estimations as well as sub-query size ($1/p$) to predict the execution time for the sub-query. We use a binary heap to maintain distances to the closest node from each query point when $id = 0$. The heap functions use the distance field for ordering within the heap.

On each iteration, the server with the smallest distance d clockwise to any of the p query points is chosen, and id is set to the corresponding distance (these distances are strictly increasing). The next server is given by the function *successor* and the distance to it calculated and re-inserted in the heap. A subtle point is that by maintaining absolute distances (i.e. always assume $id = 0$ and compute distances to the corresponding i/p) we do not have to update the distances in the heap when the id changes.

The algorithm stores the currently best query delay and best id. It also keeps the delay of the current server configuration. When we switch one server with another, we compute the delay of the new server; if it is greater than the current delay, we re-set the current delay. If, however, the new delay is smaller than the current query delay AND the node being replaced was the slowest node, we iterate all server delays and recompute the max (this last part iterates over all p so it is slow; this is why the algorithm tries to avoid it when possible).

Algorithm 1 ROAR Scheduling Algorithm

$delay_q \leftarrow 0$
for $i = 0 \ldots p-1$ **do**
  $assigned[i] \leftarrow node\_in\_charge(i/p)$
  $finish[i] \leftarrow estimate\_finish(assigned[i], 1/p)$
  if $delay_q < finish[i]$ **then**
    $delay_q \leftarrow finish[i]$
  end if
  $d.distance \leftarrow assigned[i] - i/p$
  $d.pos \leftarrow i$
  $insert\_heap(d)$
end for
$delay_{best} = delay_q$
$id_{best} = 0$
$id = 0$
while $id < 1/p$ **do**
  $d \leftarrow remove\_heap()$
  $id \leftarrow d.distance + 1$
  $assigned[d.pos] \leftarrow succesor(assigned[d.pos])$
  $isMax \leftarrow finish[d.pos] == delay_q$
  $finish[d.pos] \leftarrow estimate\_finish(assigned[d.pos], 1/p)$
  if $isMax$ **and** $finish[d.pos] < delay_q$ **then**
    $delay_q = \max(finish)$
  else
    if $finish[d.pos] > delay_q$ **then**
      $delay_q = finish[d.pos]$
    end if
  end if
  if $delay_q < delay_{best}$ **then**
    $id_{best} \leftarrow id$
    $delay_{best} \leftarrow delay_q$
  end if
  $d.distance \leftarrow assigned[d.pos] - d.pos/p$
  $insert\_heap(d)$
end while

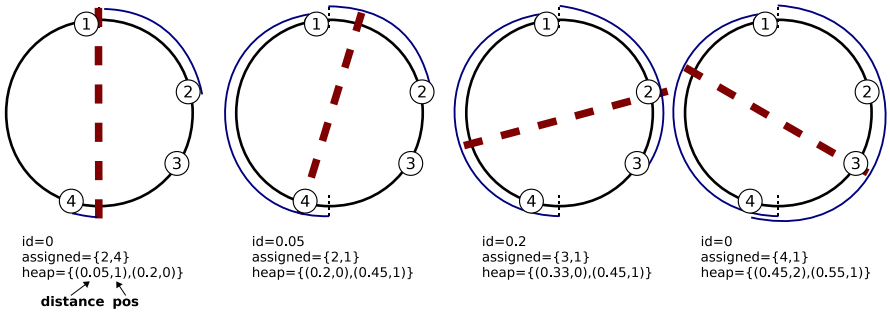


Figure 4.5: ROAR Scheduling Algorithm: Simple Example

An example is provided in Figure 4.5, with four nodes and $p = 2$; let the circle length be 1. The dashed line intersects the ring at the two query points (p_1 and p_2). The parallel blue arcs show the distances maintained in the heap. For simplicity, the *assigned* array contains the user-friendly node identifiers (1-4) rather then their positions on the ring (0.2, 0.33, 0.55, 0.95).

The algorithm starts with $id = 0$ and servers 2 and 4 are assigned the respective sub queries. In the heap distances are maintained to server 4, and server 2. Next, the id is increased past server 4; now servers 2 and 1 run sub-queries, and the heap is updated to include the distance to node 1. Note that the distance does not depend on the current value of id, being computed relative to positions 0 and 0.5 on the ring. The next step is to increase id to 0.33, past server 3; now servers 4 and 1 run the query. The next step would pass node 1, selecting the starting configuration with servers 2 and 4 running the query. At this point id is close to $1/p$ (i.e. 1/2) and the algorithm finishes.

The complexity of the algorithm is $O(n \log p)$. n is given by the number of iterations: we have one step per node in the system. $\log p$ comes from removing the closest server on each iteration from the heap, and adding the new server. Finally, we show experimentally that the $O(p)$ required when we are replacing the slowest server with a faster one is amortised over the n iterations.

In comparison, our straw-man deterministic algorithm has complexity $O(np)$. In practice this difference matters: if $p \sim n \sim 1000$, our algorithm is 100 times faster.

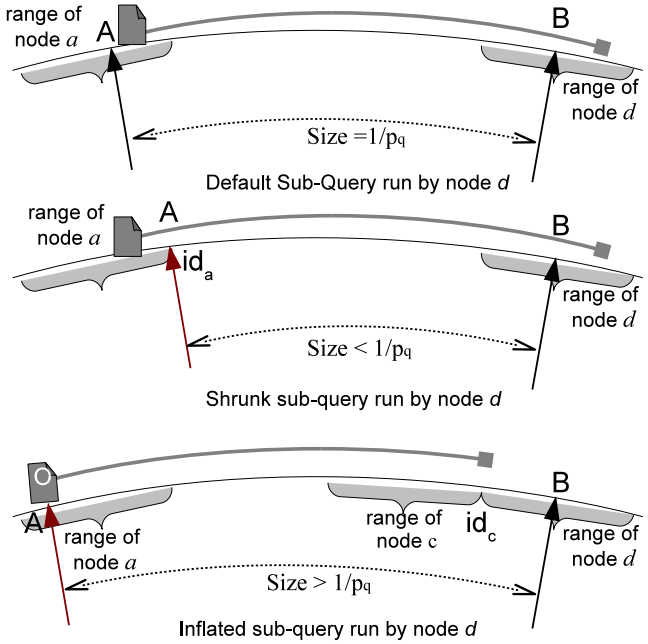
Scheduling for the PTN algorithm is simpler. For each sub-query, the front-end will iterate through all the servers in a cluster. Together, the complexity is $O(n)$. For the practical example above, we found that ROAR scheduling is 3 times slower than PTN, taking 20ms instead of 8.5ms.

Scheduling for Multiple Rings. It is straightforward to extend the above algorithm for multiple rings. Two things will change: first, when computing the assignment and finish time for a sub-query, the algorithm will consider both rings and use the fastest server. Second, when searching the successor of a node to update the distances, it will consider nodes from both rings, effectively *overlaying* node identifiers from both rings. The complexity of the algorithm remains the same.

4.8.2 Optimisations

Range Adjustments. The scheduling algorithm and all our previous discussion of ROAR (except dealing with failures) assume sub-queries have equal size. Their sizes is dictated by the smallest p than the system is currently configured to support. If we chose bigger sub-queries there will for sure be one or a

Figure 4.6: Range Adjustment for Query Scheduling



few nodes who would not be able to correctly run their part.

However, we make the observation that we can increase the length of some sub-queries while still allowing correct execution. The reason for this is the fact that ROAR over-replicates in some cases, when object replication ranges briefly intersect node ranges.

Figure 4.6 illustrates this concept. Nodes a and d run two consecutive sub-queries. It is always safe to reduce node d's sub-query by moving point A to the right: all the objects with id greater than A are already replicated onto a, as long as $A < id_a$.

We can also increase the sub-query allocated to node d by moving A to the left. The constraint here is that $A + 1/p_q > id_c$; in other words, we can shift A left as long as the objects with id greater than A are replicated on d.

We use this technique to take work away from the node that finishes last and push it to its neighbours. The aim is to equalise the finishing time across the neighbours, as long as the above two constraints are met.

The algorithm is very simple, taking near constant time. We experimentally show it is most effective when the replication level is low, making node ranges and sub-query sizes comparable in size.

Increasing the Number of Sub-Queries. Query delay is dictated by the slowest server to finish running its sub-query. While scheduling, the front-end knows which sub-query will be late to finish, potentially delaying the whole query. To avoid this, the front-end can dynamically split the slow sub-query and allocate it to faster nodes, with a technique is similar to the one used to deal with failures. This process can be repeated, with the front-end always selecting the slowest sub-query, splitting it and allocating each sub-query to the fastest servers that can run them.

For a given value of p and a fixed starting point of the query, tasks of size $1/p$ can be run by a single server in the system. For the same p, each half size sub-query ($1/2p$) can be run by as many as r servers. Hence sub-query splitting not only reduces the load of the slowest node, but also offers numerous alternatives for sub-query placement.

In contrast to the range adjustment optimisation presented above, this optimisation increases fixed overheads associated to each query: the more we split, the more messages the front-end needs to send, the more query threads are started on the ROAR servers, etc.

We show in the analysis section that most of the benefits come from splitting a single sub-query, so the costs of using this technique may be worthwhile. Another way to reap most benefits without paying the costs is to use it only when the slowest server is significantly slower than the others.

4.8.3 Multiple Front-End Servers

Although the scheduler is centralised, our experimental analysis shows that a machine can support thousands of servers at high query rates. However, it is important to be able to use multiple front-end servers for fault tolerance, and to further increase scalability.

If fault-tolerance is the only concern, it is straightforward to maintain a backup front-end server, pushing the relatively rare long-term topology changes to both master and backup servers. It is not necessary to push other state—such as server processing speeds, or liveness information— to the backup. The latter will quickly learn all this information when it comes online, providing little if any disruption to query delays.

The value of p should be kept updated on the backup, but this is an optimisation rather than a requirement. If the backup does not know what value of p is safe to use it can either start using $p = n$ (which will always work) and progressively decrease p. Another option is guess a value of p and use it to split queries. If the servers do not have enough replicas they will reply saying they haven't matched the whole query. Then, the front-end can decrease p and retry.

It is easy to use multiple front-end servers in parallel. The exact behaviour depends on how query delays vary with the number of concurrent tasks. Memory and CPU-bound query processing will typically run t concurrent tasks t times slower; in such cases, the front-end schedulers can schedule queries independently, in a completely decoupled fashion. To avoid oscillations in server processing power estimates and in query allocations, statistics about servers should be averaged over many queries. The same applies for disk-bound query processing, assuming reads are big enough to avoid disk head thrashing.

4.8.4 Sending Queries Reliably

The front-end needs to reliably send sub-queries from the front-end to the ROAR servers and to carry back the results. TCP is the obvious transport protocol to use as it offers reliable delivery, has stable implementations and a well known API.

Yet standard TCP suffers long timeouts when the connection is application limited. The queries are small, so at any time there is little data in flight between the scheduler and any of the ROAR servers. If a packet gets lost, fast-retransmit is not triggered; instead, a long retransmit timeout [4] must expire before

[4] The TCP standard suggests setting the minimum RTO to 1 second. Most OSes set it to smaller values. Linux uses 200ms.

the query is re-sent. By that time most of the query results may be already received at the front-end, and the scheduler may reschedule the missing query onto another server. Retransmitting the query is useless, yet TCP must send its outstanding data to function properly. This is the head-of-line blocking problem: the controller cannot schedule a new query until the old one is needlessly executed.

When p is small enough and the network is relatively idle, packet losses are very rare so this is not an issue. However, when p grows large (say 1000) we have p servers replying to the front-end at roughly the same time. Such synchronisation overflows the switch buffer on the link to the front-end (this is called the TCP incast problem [CGL$^+$09, VPS$^+$09]). To make matters worse, even retransmissions after timeouts may be synchronized, causing further loss.

A very simple fix is to drastically reduce or even eliminate TCP's min RTO bound, as proposed in [CGL$^+$09, VPS$^+$09]. In this way, retransmissions will happen after a few ms, and most of the problems above vanish. There is still head-of-line blocking, however on much shorter timescales (ms). As query delays are on the order of tens and hundreds of ms, blocking for a few ms is not an issue.

If it were an issue, we could use UDP enhanced with application-level acknowledgements, but the difficulty is to avoid congestion collapse in pathological cases. A better choice would be to use DCCP [KHF06] that provides congestion control without mandating reliable transmission (thus eliminating head-of-line blocking).

4.9 Managing Ring Membership

We have discussed at a high level how nodes are added and removed in ROAR. Here, we describe our practical instantiation of these ideas, and provide further details on how ROAR membership works in practice.

We use a centralised membership server to keep track of nodes' assigned ranges, and to ensure that the system is load balanced. The membership server downloads periodic statistics from the front-end servers about node liveness and processing speed. It uses all this information to:

- Insert new servers at hotspots.
- Enable or disable server local load balancing
- Decide when to move servers to different parts of the ring or even across different rings
- Redistribute the failed node's range between its neighbours when long-term failures are detected.

The membership server can be configured to organise servers in one or more rings. It attempts to give equal processing capacity to each ring, as this gives the best query delay (as we show in the Analysis section).

When a new server joins, the default behaviour is to pick the ring with least processing capacity and to add the server into the hottest spot of that ring. The membership server does not utilise individual server load estimates to decide how to allocate ranges, as these can be skewed by the front-end's preference for fast servers when allocating sub-queries. Instead, it uses the ratio of range to processing power

as a proxy for the load of that node. The front-end will produce such an allocation only when the system load nears 100%.

Once a node is given a range, it will start downloading the required objects from the backend file-store. As it completes all objects for the range (or a part of it) it informs the membership server. At this point, the membership server marks the server as up and records its available range. This information is then pushed to the front-end servers, which will start scheduling queries on this node. When shrinking a node's range, the membership server first shrinks its recorded range, updates the front-end, and only then tells the node to shrink its range.

As we have mentioned, the ROAR servers perform local load-balancing independently, periodically announcing their new ranges to the membership server. To avoid churn, we set a threshold on the load difference between nodes (10% for our implementation): if the difference is less, the nodes stop balancing the ranges. The membership server can disable local range balancing if desired; this is done by pushing a range update to the corresponding nodes with a "Fixed" flag. This is to allow administrators to tweak node ranges as desired.

Further, local load-balancing can take a long time to balance if one area of the ring is really "hot" and the opposite area is "cool": pairwise range changes propagating out of the hot area will create a lot of unneeded object churn and will take a long time to load balance. The membership server has a global view of the ring and will simply move nodes from "cool" places of the ring to the hot ones, significantly speeding up this process.

The membership server maintains a history of range allocations to servers. If a server is taken out for maintenance and brought back up it will get the same range it had before; it only needs to download deltas in its object list since it was previously online.

Finally, as with the front-end server, the membership server can be replicated for availability purposes, with only one master server active at any point in time.

We have discussed the basic operation of the membership server; we now look at how it can be used to optimise for common data centers that have daily load fluctuations, and how it can reduce cross-sectional bandwidth usage.

4.9.1 Adapting to Changing Load

Most online services see fluctuating load with diurnal and weekly patterns [CHL+08, CFSS05, WAB+06]. The ratio between the mean load in different parts of the day or week is 2x to 4x. A service provider could keep all of their nodes up all the time, but that would waste energy. It is better either to turn off some of these servers when load is low, or to use them for other tasks [CHL+08].

If ROAR uses a single ring, it can shut servers down in a pattern that does not dangerously reduce the number of replicas in any part of the ring, hence maintaining high system availability. However, the ring will be now fragmented, and the number of choices for any query will be even less than the r choices in SW. It makes more sense to use multiple rings for such scenarios, and turn off entire rings when needed.

We have already mentioned how the rings are populated. Say ROAR has been configured to use 4

rings, with each ring maintaining two replicas of the data. The membership server will use load statistics provided by the front-end server to decide how many rings it should have running at any given point in time. The system can easily bring some of the rings online or shut them down to track the average load, and to match the predicted future load. The time needed to bring a ring online is of concern. The membership server assigns the same node ranges to returning servers, so start-up delay can be minimised if the same servers are periodically shut-down and brought back up.

4.9.2 Reducing 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 partitioning in cross-sectional bandwidth usage.

Assume that object replication is much more expensive - bandwidth wise - than running queries. This is true for most distributed search applications we have analysed. The case when queries are bandwidth hungry can be optimised in a similar way.

PTN 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, minimising cross-sectional bandwidth consumption.

ROAR can similarly use physical placement of servers to minimise update cost, by having the membership server assign 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 PTN.

To implement this optimisation in ROAR, it suffices to use the peer-to-peer like object 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.

The downside of such server placement in both ROAR and PTN is vulnerability to correlated server failures in the same rack. These are not all that unlikely: if the top-of-the-rack server fails, or the rack's power supply burns out, the whole rack is wiped out. Finally, our back-of-the-envelope estimations for web search in the analysis section show cross-sectional bandwidth usage is not of concern.

Chapter 5

Application: Privacy Preserving Search

Online storage of personal data (such as videos, photos, documents) is now becoming commonplace. However, data are typically stored as "plaintext" which makes it easy for online companies, law enforcement agencies and hackers to access users' data without them knowing it. Serious privacy concerns have already been raised by the Federal Trade Comission [Tec10]. To protect their privacy, users could in theory encrypt data before storing them online. The downside is that accessing the data becomes much more difficult. In particular, searching is not directly possible.

In this chapter we present Privacy Preserving Search, a search application well suited for paralelization with ROAR. With Privacy Preserving Search the client stores encrypted metadata on the server(s), describing photos, documents, pictures, email messages, and so on. The client then creates encrypted queries and gives them to the server. Privacy Preserving Search techniques allow the server to select the encrypted metadata that matches the query without knowing the contents of the query or the metadata. These are returned to the client, which decrypts them.

We begin by describing the motivation for Privacy Preserving Search and focus on the techniques that make it possible, targetting common query types appearing in practice: keyword and numeric matching. We present a novel construction to support numeric matching and to rank query results. Finally we analyze the scaling bottlenecks PPS faces and discuss how to paralelize PPS with ROAR.

5.1 Motivation

We are witnessing a compelling shift towards what is called an "online" operating system. While online email has been around for quite some time (e.g. Gmail, Yahoo mail, Hotmail), other parts of the users' desktop are being shifted online as we speak: documents (Google Docs, Microsoft Office Live), pictures (Picassa, Flickr), videos (YouTube). It seems that the part of the local hard drive that contains personal data is moving online.

The main benefits to end-users are easy sharing, availability and accessibility: personal files are now always online and can be accessed from anywhere by just using a web browser. Further, they are guaranteed to last: the online providers use massive redundancy, and anything short of a large scale disaster will likely not affect their durability. This is not true for files on users' hard drives, where a failure can make years' worth of personal data disappear.

Service providers such as Google aggregate and store users' private data including documents, videos, photos, email, friends lists, browsing history, search history, and so on. This entails higher privacy risks: the user has little or no control over who accesses its data and when. The Federal Trade Commission has recently brought up these issues, pointing out the increased privacy risks of online data storage: "the ability of cloud computing services to collect and centrally store increasing amounts of consumer data, combined with the ease with which such centrally stored data may be shared with others, create a risk that larger amounts of data may be used by entities in ways not originally intended or understood by consumers" [Tec10].

If this data were only stored locally, on the users' devices, all these privacy risks would be much smaller. Ideally, we would like to have the same privacy for our online files as if they resided on our own devices (assuming these are secure).

The basic recipe to protect the privacy of user data in the cloud is simple. Users should symmetrically encrypt their files using their private key before storing them on the servers. To go further and even hide the file size, files could be broken into blocks and stored as such. When reading, several blocks would be used to compose larger files. In effect, the online providers would offer a simple block storage device, that users would use to store and retrieve their files. This is similar to the S3 service already offered by Amazon.

This storage service underlies most online services, and seems enough for any application if the server is not needed to implement other functionality. Instead of using the local hard drive, the software running on the user's machine will download/decrypt necessary files before using them and encrypt/upload them afterwards. One exception is securing email, where messages are created by other users. There, public key cryptography could be used instead for message encryption.

When the number of files becomes large, it becomes cumbersome for users to find information of interest. Traditional file system hierarchies help to some extent. Search is the missing ingredient as it is user friendly, faster and more powerful especially on portable devices like mobile phones. Search has become ubiquitous in accessing web and local information, so it is likely it will be central in providing an agreeable user experience. The success of Apple's Spotlight search service in Mac OS X is representative of this new trend for quick access to information. Search obviates the need for deep, cumbersome hierarchies of directories and folders. Fast search is a requirement if privacy preserving online applications are to become successful.

5.1.1 Limitations of Online Privacy

We acknowledge that certain online features are difficult to implement in a privacy preserving manner. These include converting files, image or video editing (e.g. changing sizes for images, bit-rates for videos), etc. While it is convenient to have this functionality online, we observe that the same functionality can be run on any home machine, given the appropriate software. Thus, if privacy is important, running the software locally is the option.

Encryption significantly increases user privacy, but is not perfect. There are fundamental privacy limitations given by the fact that the servers store user data. Servers analysing client requests will be

able to infer which blocks are likely to be part of the same file and which files are important at any given point in time.

5.2 Basic Approach and Scope

Privacy preserving search is the main focus of this work, and it has two types of solutions. The obvious solution is to create and maintain an index of files on the servers, downloading it before queries and uploading it after files change. The second solution is to perform the search on the servers themselves, using encrypted queries ran against encrypted metadata describing the files. In the latter case, when a file changes, its corresponding metadata is updated.

To guide the comparison of these solutions, let us examine the typical deployment environment for PPS. Today most users use personal computers to access their online data. However, there is a strong trend towards integrating more and more functionality on mobile devices, so these may well be the gateway to the user's files in the near future. Further, the number of devices each user owns is increasing: typical users have a work computer, a home computer and possibly a laptop, a mobile device, a portable music player, and so forth.

Device lifetimes vary wildly, with portable devices typically having a much smaller lifetime than their desktop counterparts (because they break more easily, are stolen, or simply because devices with new desirable features appear). Thus, it seems a bad design decision to place the focus on a unique piece of equipment, and even to assume a user's device set will stay constant. Online storage ensures an easy transition between devices, and is effectively the only long lived device the user owns. A design requirement is that all these devices should be able to seamlessly access and search the online repository.

Among these devices network connection speeds vary significantly (from 1Gbps/s for Gigabit Ethernet to as low as 28.8Kb/s for GPRS) and so do costs: wired Internet connections typically have a flat rate and unlimited traffic, while mobile connections have a monthly quota and volume charging when the quota is exceeded. Further, battery for mobile devices drains quickly when sending or receiving data. Another design requirement is to minimise bandwidth usage to reduce costs and increase battery life on mobile devices.

Finally, while the main application of privacy preserving search is to allow users search their online repositories of files, additional uses also seem plausible. Push-based notifications are very useful as they provide the users the ability to create filters and install them on the servers to be notified when certain events occur.

5.3 Analysis of the Index-Based Solution

The simplest way to implement the index-based solution is to update the encrypted index on each file change, and to download the index whenever it changes. This unnecessarily wastes bandwidth for both updates and queries. A better algorithm is to encode and encrypt each index change as a delta to the index and store it online separately. When the user runs a query, it typically needs to download only the latest deltas instead of the whole index; it locally applies the deltas to the index and only then runs the search locally. As the deltas themselves can become numerous, the index is updated to include all the

deltas periodically or when a threshold number of deltas have been created.

The index-based solution is simple: no additional mechanism is needed at the servers for implementing it. It is well suited for users that mostly use a single device, but behaves poorly when users have many devices and have to frequently download the index.

The index-based solution does not work well for pushing notifications. Filters like "notify me when this file is updated", "when somebody sends a message containing URGENT in the title" seem very useful to prompt the user's attention when rare events happen. The alternative in the index-based solution is to periodically check the index, or to be notified when updates are added to the index. Both approaches waste bandwidth.

As we have noted, wasted bandwidth may be acceptable in an Internet setting where pricing is mostly flat, yet it entails high costs and decreases battery lifetime for mobile users.

5.3.1 Bandwidth Comparison

We performed a simple comparison of bandwidth usage in the index-based solution and in our encrypted search solution (presented later). We assumed the online storage contains 50.000 files [1]. We create a simple index by listing all the filenames in a text file, which we compress and encrypt. The size of the output is 500KB, requiring around 10B per file. Updates to this index are encoded as filename and the change (added, deleted, updated). Compression is less efficient on updates; one update, compressed and encrypted, takes 200B. In our privacy preserving search implementation, we create one metadata for each file. The size of the metadata is 500B; a single encrypted query also takes 500B.

We built a simple analytical model of bandwidth usage by both algorithms. The bandwidth used by PPS is $500f_u + 2500f_q$, where the first term accounts for the frequency of updates and the second term account for the frequency of queries, and assumes only 10 results of 200B each are returned.

To approximate the bandwidth used by the index-based approach we proceed in two steps. Let the maximum number of deltas be δ_{max}.

First, the expected bandwidth use due to updates is $f_u(500.000+200(\delta_{max}-1))\frac{1}{\delta_{max}}$. The formula reflects the fact that in δ_{max} updates, the index is stored once completely, and $\delta_{max}-1$ updates are sent.

To compute the bandwidth used in queries, lets first assume updates are generated on another computer than the one that does the search. Before each search, the computer checks the online version of the index and downloads the index, deltas or both depending on the local version. To begin, let the frequency of queries f_q be smaller to the frequency of updates f_u. Depending when the query arrives, the querying computer will download the index, one delta, two deltas, up to $\delta_{max}-1$ deltas. Assuming these are equally likely, the expected query bandwidth usage is $f_q(500000+100\delta_{max}(\delta_{max}-1))\frac{1}{\delta_{max}}$.

The value of δ_{max} that minimises bandwidth consumption depends on both f_q and f_u. We compute the optimal value and plot the ratio of bandwidth consumption in the index-based approach to the bandwidth used in PPS, varying query and update frequencies.

In general we expect the number of updates to be larger than the number of queries, but do not

[1] This was the number of files in the authors' home directory four years ago; now the same directory has grown to nearly a million entries

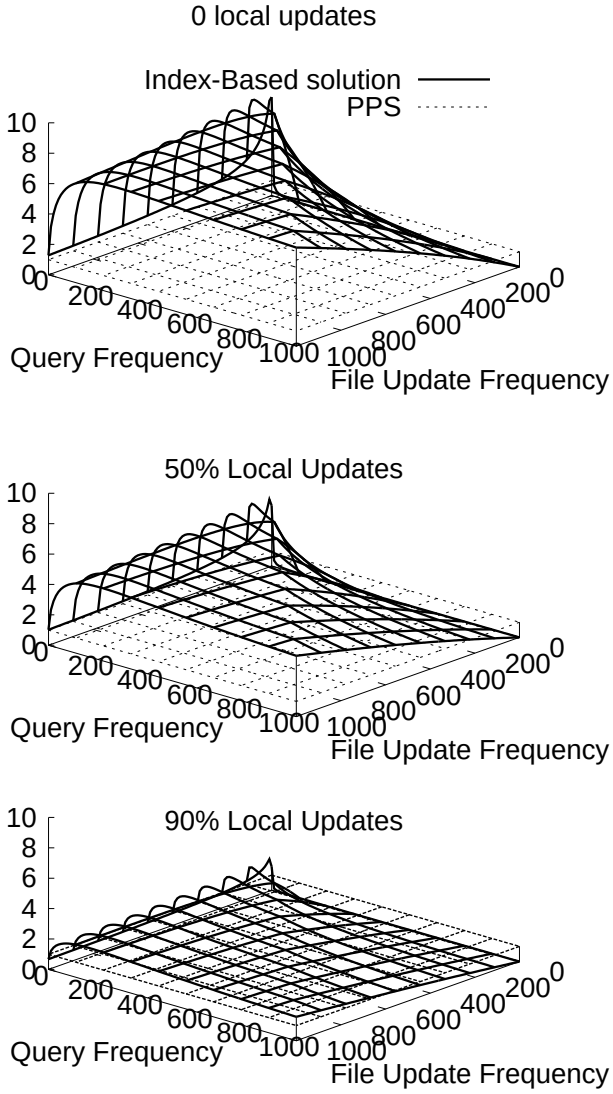


Figure 5.1: Bandwidth Consumption Comparison between Index-Based solution and PPS

restrict the analysis to this case. When the query frequency is larger than the update frequency, we modify the formula above to use the update frequency when computing bandwidth required for queries.

We vary both update and query frequencies from 1 to 1000 and consider three cases: one where all the updates are generated by another machine, one where half of the updates are local (and do not need downloading) and one where 90% of updates are local. In figure 5.1 we plot the relative bandwidth usage of the index-based solution when compared to the PPS solution.

The results are not surprising, and they show that the index-based solution consumes more bandwidth overall, as it generates eight times more bandwidth when updates are non-local, and nearly twice more traffic more when most updates are local.

For mobile devices, this has several implications. First, bandwidth costs will be significantly higher. Secondly, the time required to run a query will be higher too. In the case where the index is 500KB,

downloading it using state of the art 3G connection running at 1Mb/s takes around 5s. This is too slow to be usable, and it is bound to get worse with more files. Download times scale linearly with the size of the index, so the more files we have, the longer it takes to search them. Finally, more traffic significantly reduces the battery life of the phone, because the wireless interfaces are quite energy-hungry [SNR+10, HGSW10].

In summary, the index-based approach has high worst-case delays, does not scale well with the number of files, and consumes a lot more wide-area bandwidth than PPS. Additionally, it is difficult to apply to online email, and cannot properly support asynchronous notifications.

The PPS solution we propose lowers wide-area bandwidth costs, but requires significant processing support in a data center. We will show during the course of this thesis that it is possible to keep search times very low for a wide range of objects searched.

5.4 Definition of Privacy Preserving Search

We wish to allow an untrusted third party, the online server, to match encrypted queries against encrypted data provided by a user. We assume the user has a private key it uses to encrypts both queries and (meta)data.

The real file data is not encrypted with the algorithms we describe below, but rather with a traditional symmetric encryption algorithm such as AES [DR02]. The metadata describing the file (also referred to as data, for simplicity, during this thesis) is encrypted and attached to the original symmetrically encrypted file, such that the server can return the matching file if requested.

We take the view that not only files can be stored on the server, but also long-standing queries (also called queries). When (meta)data is added, modified or deleted on the server (by the user), the server will match the new metadata against the standing queries and notify the user if it matches. We conflate the two mechanisms as together they offer all the functionality needed by the user.

5.4.1 Security Preliminaries

We say that a function f is *negligible* in t if, for any polynomial p there exists t_0 such that for all $t > t_0$, $f(t) < 1/p(t)$. We use PPT as a shorthand for *probabilistic polynomial time*.

We provide the following standard definitions from the literature on provable security [Gol01], which we will use throughout this chapter.

Pseudorandom Function. A pseudorandom function is computationally indistinguishable from a random function. Formally, a function family $\{F_K : \{0,1\}^n \rightarrow \{0,1\}^m | K \in \{0,1\}^t\}$ is pseudorandom if for every PPT oracle algorithm A the following value is negligible in t: $|Pr[A^{F_K(\cdot)}(1^t) = 1] - Pr[A^R(1^t) = 1]|$, where R is a random function selected uniformly at random from the set of functions from $\{0,1\}^n \rightarrow \{0,1\}^m$. The probabilities are taken over the choice of K and R, respectively.

Pseudorandom Permutation. A pseudorandom permutation is computationally indistinguishable from a truly random permutation. Formally, a permutation family $\{E_K : \{0,1\}^n \rightarrow \{0,1\}^n | K \in \{0,1\}^t\}$ is pseudorandom if for every PPT oracle algorithm A, the following value is negligible in t: $|Pr[A^{E_K(\cdot)}(1^t) = 1] - Pr[A^\pi(1^t) = 1]|$, where π is a permutation selected uniformly at random from

the set of bijections from $\{0,1\}^n \rightarrow \{0,1\}^n$. The probabilities are taken over the choice of K and π, respectively.

5.4.2 Security Assumptions and Scope

We use the term "user" to describe a number of different devices, possibly belonging to different people, which were authorised—by being given either the secret key or an encrypted query— to search the system. We assume that each of these devices is trustworthy. How the key is shared between these devices is orthogonal to the privacy preserving protocol; typically a smart card could be used.

We assume that servers are computationally bounded and do not deviate from the privacy preserving search protocol—they correctly return matching files to the user. Otherwise, denial of service attacks could be mounted easily, affecting the correct operation of the infrastructure. If we bear in mind that these services are in some way paid for by the user, it seems irrational for servers to risk losing users by denying them service.

5.4.3 Problem Definition

Definition 7 (Privacy Preserving Search (PPS))**.** Consider a user U that stores a number of files on a server, and that has two types of inputs. The user generates a sequence of metadata, with one metadata describing one file at any given point in time. The user also generates queries, a subset of which may be active at any point in time. PPS is a multi-round protocol between U and a third party S, the server. In each round one of the following can take place: a) U submits a one time query to S; b) U submits or withdraws a long standing query to S; or c) U sends a metadata to R. A correct implementation of PPS with security parameter t must satisfy the following:

1. **Correctness.** S must be able to determine in PPT the subset of long standing queries that match new metadata, and S must be able to find the set of metadata matching a one-time query.

2. **Security.** For $k \in \mathbb{N}$, define $View_k$ as all the communications S has received from U before round k. Define $Plaintext_k = \{M_1, \ldots, M_i, Q_1, \ldots, Q_j\}$ as the set of metadata and queries from U before round k.

   Let O_k be an oracle that has access to $Plaintext_k$ and exports the two following functions: *match*(idx_Q,idx_M), defined iff $idx_Q \in \{1, \ldots, j\}$ and $idx_M \in \{1, \ldots, i\}$ that answers whether the query denoted by idx_Q matches the metadata idx_M. , and

   coveridx_{Q_1}, idx_{Q_2}, where the indices are defined in $\{1, \ldots, j\}$, which replies with yes or no to indicate whether query idx_{Q_1} covers query idx_{Q_2}. A query Q_1 covers the query Q_2 if the metadata matched by Q_1 are always a superset of the metadata matched by Q_2.

   Finally, define $View_k^* = \{i, j, O_k\}$.

   A PPS scheme is secure if, for $k \in \mathbb{N}$, for any PPT algorithm A, any function h, there exists a PPT algorithm A^* such that the following value is negligible in t: $|Pr[A(View_k, 1^t) = h(Plaintext_k)] - Pr[A^*(View_k^*, 1^t) = h(Plaintext_k)]|$

In other words, we require that information leaked to the server is the same as in an ideal protocol where the server performs its functionality by submitting the indexes of the queries and metadata it wishes to match (idx_Q and idx_M) to an oracle (O_k) with access to the plaintext versions. The above definition implies the following:

- **Metadata Security.** Metadata encryption is semantically secure for multiple messages (as defined by Goldreich [Gol01]) in the absence of queries. When queries are available, the only thing that is leaked is whether a metadata matches the query or not. The metadata that are not matched by the available queries are computationally indistinguishable from random bits.

- **Query Security.** Queries can be distinguished with the covering relation, and therefore their encryption scheme is not semantically secure. A stronger security model could require that the query encryption scheme is also semantically secure. In this work we discard this stronger model for practical purposes: efficient solutions for executing continuous queries rely on the coverage relation between queries, which mandates that a server should know if two queries are related [CW03].

  Further, even in the one-time query scenario, when a a query is run against a large number of metadata and returns a non-trivial number of results (i.e. not zero, and not all the metadata), one can strongly infer that queries that return exactly the same results (although the results are encrypted) are equal. By acknowledging that in practice it is very difficult to obtain query indistinguishability we are able to obtain more practical solutions.

- **Metadata Unforgeability.** It is infeasible for an adversary to create valid encrypted metadata. This is important, since an adversary able to craft arbitrary metadata can use regression techniques to infer an approximation of the query function.

- **Query Unforgeability.** It is infeasible for an adversary to create valid encrypted queries. Otherwise, the adversary can use binary search to discover the value of the metadata in logarithmic time. An important consequence of query and metadata un-forgeability is that plaintext queries or metadata cannot be used in the matching process (since these are easy to create by adversaries).

- **Match Isolation.** It is infeasible to compute anything from the messages seen at the server that cannot be computed by applying *match* and *cover* (using an oracle) to the indexes of queries and metadata.

The definition above can be generalised naturally to the multi-server case where the number of matching servers is arbitrarily large.

Any solution for PPS consists of the following five algorithms, the first four being required and the fifth optional:

`Keygen(`t`)`**:** run by the user, U outputs the private key K when given the security parameter t as input

`EncryptQuery(`K`,`Q`)`**:** run by the user, outputs the encrypted query Q_e when given the plaintext query Q and the private key K

`EncryptMetadata(`K`,`M`)`**:** run by the user, outputs the encrypted metadata M_e when given the metadata Q and the private key K

`Match(`M_e`,`Q_e`)`**:** run by the server, receives as parameters an encrypted metadata M_e and an encrypted query Q_e and outputs 1 if Q_e matches M_e or 0 otherwise

`Cover(`Q_1`,`Q_2`)`**:** run by the server, receives as parameters two encrypted queries Q_1 and Q_2 and outputs 1 if Q_1 covers Q_2 or 0 otherwise

For simplicity of exposition, we use the term "encrypt" to denote a secure encoding of queries and metadata that allows PPS. However, we point out that the schemes presented here are not traditional symmetric encryption schemes, since decryption is not usually possible.

5.4.4 Limitations of Confidentiality

Regardless of the protocols used, the maximum level of attainable confidentiality in PPS is quite limited. These limitations arise from the functionality the server is required to perform (i.e., to decide if an encrypted query matches an encrypted metadata) and are inherent to the PPS problem. Here, we present a brief overview of these limitations.

Limited Metadata Indistinguishability

Queries stored by a server can be used to distinguish certain metadata (e.g., to tell if they are equal) by matching the queries against the metadata: this uses the fact that the server must be able to match queries against metadata, and is independent of the encryption scheme used for metadata. The more queries that are available, the more likely the server is to accurately distinguish metadata. In the case where the server has a complete basis of queries, it can distinguish all metadata with zero probability of error.

Statistical Attacks

The server can find for each query it runs how many of the files match. If the server has additional information about the corpus it can infer with some degree of confidence what the search term could be by looking at the number of matches and the frequency of the search.

While this problem is significant for general purpose web-search queries, we believe that when a user searches its own files the file content and the searches will vary significantly across users, and therefore may yield limited information without specific profiling for the actual user.

Confidentiality-Generality Tradeoff

We define the *complexity* of a query type as $\frac{1}{min_S}$, where min_S is the minimal number of queries needed to recognise all metadata. There is a direct correlation between the complexity of a query and the information it leaks about metadata. For instance, the simplest query function is equality testing: one such query will allow a server to distinguish metadata that are equal to the specified value. To distinguish all possible metadata without error (i.e., to have a basis), the server needs $O(2^n)$ distinct queries, where n is the size of the metadata in bits. The more complex queries are, the more information is leaked about metadata. For instance, a query that accepts all metadata with the k^{th} bit set to a specific value, will

allow the server to distinguish information about the k^{th} bit of all metadata. In this case, only $O(n)$ queries are needed to distinguish metadata with zero probability of error.

5.5 Solutions for Privacy Preserving Search

What features of documents should be searchable? Traditional filesystems provide tools like "find" and "grep" to match through filenames, file attributes and file content. Consequently, we split searchable file information in three searchable attributes: file name and path, file content and attributes like size and modification date.

To support practical searches on file name and file content we need to support keyword matching: the ability to tell if a keyword is contained in a collection of keywords. For filename matching keyword matching should suffice, and clearly all the components of a path must be searchable. For document content, on the other hand, it makes little sense to include all the keywords contained in each document in the searchable attribute: when one searches for the keyword "the" all the documents written in English would be returned. Basic information retrieval techniques only index the most important features of each document, i.e. the words with the highest discriminating power. Thus, we imagine that for each document a small number of keywords will be searchable (say 50).

To support matching on filesize and access dates we must support number matching: the ability to tell if a given encrypted number lies within a range.

Combining keyword matching and number matching, we can enhance the precision of content search as follows. Assume each keyword is ranked based on its importance in the document; the ability to search for documents where a certain keyword is the most important feature, or in the first 10 most important features, allows us to indirectly obtain ranked results.

In this section, we present PPS algorithms for these tasks. We identify in the literature two algorithms that support basic keyword matching from Goh [Goh03a] and Chang et al. [CM05a] and list them here, showing they both conform to our security definition. Using these as building blocks, we present novel constructions that allow PPS number matching and ranking of search results. We also discuss techniques to support more general queries.

In the descriptions of the basic PPS schemes, we assume each metadata is a single value, and each query is a single predicate. Because all of the interesting protocols we describe below are based on keyword matching, all the different types of file information can be easily bundled into a single searchable "attribute" at the server; we describe how we implement this in section 5.6.

Practical queries may involve multiple keywords and may refer to file attributes such as last modification date. Ideally, we would like to "compose" all these predicates into a single query which the server runs. However, as we will see in our discussion in section 5.5.5, this is quite expensive for two-keyword queries and prohibitively so for general purpose multi-predicate queries.

We use a less secure but practical alternative, encoding predicates separately and having the server compose them. With this scheme the server gains more information than necessary. For instance, a query requesting keywords A and B will reveal to the server which metadata match A and which metadata match B, instead of only revealing the metadata that match both A and B.

5.5.1 Equality Matching

We begin by showing how to support simple equal filtering of attributes. Although this scheme is not powerful enough to be used in practice, it is useful as a starting point for understanding the other mechanisms.

To support equality matches, we use the first step of the solution proposed by Song et al. for searches on encrypted data [SWP00]. The idea is to compute the "hidden" value of an attribute by passing its plaintext value as argument to a pseudorandom function, keyed with the secret key. The encrypted query is the hidden value of the plaintext. Encrypted metadata are composed of two parts: a random nonce r, generated by the user, and the result of feeding r to a pseudorandom function, keyed with the hidden value of the attribute's plaintext.

Let F be a pseudorandom function. The algorithms for *Equal* PPS are:

`Keygen(`t`)`**:** select K from $\{0,1\}^t$ uniformly at random

`EncryptQuery(`K`,`Q`)`**:** return $F_K(Q)$

`EncryptMetadata(`K`,`M`)`**:** select rnd uniformly at random. Let $h = F_K(M)$. Return $(rnd, F_h(rnd))$.

`Match(`M_e`,`Q_e`)`**:** Let M_e = (rnd, two). Return 1 if $F_{Q_e}(rnd) = two$, 0 otherwise

`Cover(`Q_1`,`Q_2`)`**:** Return 1 if $Q_1 = Q_2$ (bitwise), 0 otherwise

Theorem 1. *Equal* is a correct implementation of PPS.

Proof. It is easy to see from the descriptions that all the schemes we propose correctly match queries against metadata and conservatively solve query coverage (i.e., they can give false negatives, but not false positives). We have also experimentally tested the correctness of our schemes. Henceforth, the proofs only analyse the security of the proposed schemes.

We want to show that for any k, any function h and any algorithm A (i.e., running at the server), there is an algorithm A^* (i.e., running with access to the oracle) such that the following value is negligible in t, the security parameter: $\delta = |Pr[A(View_k, 1^t) = h(Plaintext_k)] - Pr[A^*(View_k^*, 1^t) = h(Plaintext_k)]|$.

The idea, borrowed from Chang et al. [CM05b], is to prove that A^* can use $View_k^*$ to construct a view $View_k'$ that is computationally indistinguishable from $View_k$. If this is the case, A^* can simulate the desired functionality by calling A with parameter $View_k'$, and therefore δ is negligible.

Without loss of generality, assume that the PPS protocol consists of two consecutive phases: *registration* (consecutive rounds in which the user sends their metadata to the server) and *operational* (consecutive rounds where the user sends queries to the server). It is simple to see that if the protocol is secure in this case, it is also secure when metadata updates and queries are interleaved. Assume $Plaintext_k = \{M_1, \ldots, M_n, Q_1, \ldots, Q_k\}$, that is, the k^{th} round in the *operational* phase. Then, $View_k$ is $\{(rnd_1, f_{f_K(M_1)}(rnd_1), \ldots, (rnd_n, f_{f_K(M_n)}(rnd_n), f_K(Q_1), \ldots, f_K(Q_k)\}$.

Let us consider the special cases first. Assume $k = 0$, that is, there are no queries. A^* selects all entries in $View'_k$ (corresponding to encrypted metadata) uniformly at random. In this case, A^* simulates A properly, otherwise we can use (A, A^*) to distinguish pseudo-random bits from random bits.

Next, assume $n = 0$, meaning that no metadata have been received yet. In this case, A^* proceeds as follows. For each $i = 1 \ldots k$, check to see if there exists $j < i$ such that O_i.*cover*(j,i)=1. If such j does not exist, select query Q_i in $View'_0$ uniformly at random. Otherwise, set $Q_i = Q_j$.

A^* feeds this view to A. The only difference between $View_k$ and $View'_k$ is the way the distinct queries are chosen. We claim that whatever A can compute from $View'_k$ can also be computed using $View_k$; otherwise the pair (A, A^*) can be used to distinguish pseudo-random bits from truly random bits.

Now consider the general case. A^* generates k queries as described above and adds them to $View'_k$. Let $Q_d = Q_1, \ldots, Q_m$ be the set of independent queries. Next, A^* generates n notifications as follows.

For all $i = 1, \ldots, n$ A^* checks if there exists $j \in \{1, \ldots, k\}$ such that O_k.*match*(i,j)=1. If so, A^* generates a random nonce rnd and adds $(rnd, f_{Q_j}(rnd))$ to $View'_k$; otherwise it adds a value selected uniformly at random.

There are two differences between $View_k$ and $View'_k$: a) distinct queries are pseudo-random as opposed to truly random, and b) metadata that are not matched by the distinct queries are generated truly randomly instead of pseudo-randomly (i.e., using f). Therefore, if A can compute something more from $View_k$ we can use it to distinguish pseudo-random bits from random-bits. This concludes the proof. □

This scheme is cheap from both the computation and communication points of view. Computation-wise, the scheme adds a few cheap operations to creating queries/metadata and a single function application for matching.

5.5.2 Keyword Matching

In this section we describe two existing solutions for keyword matching that achieve similar security yet have different costs. As keyword matching will be used as a building block for our proposed protocols for matching numbers, and also for matching keywords, having to choose between two solutions with different practical characteristics gives us an optimisation dimension we can exploit.

Bloom-Filter Keyword Matching

The first protocol we use has been proposed by Goh [Goh03a]. The idea is to break the string into words and construct a Bloom predicate [Blo70] to signal existence of a word in the string. The query is a single keyword.

Let F be a pseudorandom function. Let BF be a Bloom predicate. The algorithms for Keyword PPS from Goh [Goh03a] are:

`Keygen(`t`)`**:** select r as the number of hash functions in the Bloom predicate BF with the desired false positive rate. Select K = (k_1, k_2, …,k_r) uniformly at random from $\{0,1\}^{rt}$.

`EncryptQuery(`K`,`S`)`**:** return ($F_{k_1}(S)$, …,$F_{k_r}(S)$)

`EncryptMetadata(`K`,`N`)`**:** extract keywords w_1, ..., w_n from N. Select a random nonce rnd. For $i = 1 \ldots n$, compute $(x_{i,1}, \ldots, x_{i,r})$ = `EncryptQuery`(K, w_i), compute the codeword (y_1 = $F_{rnd}(x_{i,1})$, y_2 = $F_{rnd}(x_{i,2})$, ..., y_r = $F_{rnd}(x_{i,r})$) and set $BF[y_j] = 1$ for $j = 1 \ldots r$. Return (rnd, BF)

`Match(`N_e`,`S_e`)`**:** Let S_e = (x_1, x_2, ..., x_r). Let N_e = (rnd, BF). Compute codewords y_i = $F_{rnd}(x_i)$ for i=1 ... r and check if the bit corresponding to y_i is set in BF. If there exists i such that $BF[y_i] = 0$ return 0, otherwise return 1

`Cover(`S_1`,`S_2`)`**:** Return 1 if $S_1 = S_2$, 0 otherwise.

We make the assumption that all strings have a predefined length and that they have the same number of words. This prevents an attacker from distinguishing two metadata by counting the number of bits set in the BF. When the latter assumption does not hold, we can add random bits to the BF to simulate the proper number of words [Goh03a].

Theorem 2. *Keyword* is a correct implementation of PPS.

Proof Sketch. The paper by Goh [Goh03a] presents a proof of security under the IND-CKA2 model, which focuses on document indistinguishability. Here we show that breaking PPS security for *Keyword* can be used to break IND-CKA2 security for *Keyword*, and therefore IND-CKA2 security implies PPS security for keyword matching.

The attacker in the IND-CKA2 game selects uniformly at random n distinct keywords $\{S_1, \ldots, S_n\}$ and finds out their encrypted versions by using the IND-CKA2 challenger. The attacker further selects two plaintext documents uniformly at random, N_0 and N_1, ensuring that the known keywords are contained by both N_0 and N_1 or by neither. N_0 and N_1 are passed to the challenger in the IND-CKA2 game, which replies with an encryption of N_b where b is uniformly random from $\{0, 1\}$.

Assume that the attacker can compute a functionality h given $View_1 = \{Q_1, \ldots, Q_n, N_b\}$, that cannot be computed only using $View_1^*$. If h does not depend on the value N_b, then h can compute something relating to the queries, besides the coverage relation; by using an argument similar to Theorem 1, we can see that this will allow one to distinguish random bits from pseudo-random bits, and is therefore impossible. Therefore, it must be that h depends on N_b, meaning that h will present non-negligible distinct outputs for $b = 0$ and $b = 1$. The attacker uses this output to guess the value of b, therefore winning the IND-CKA2 game. This completes our proof sketch.

Overhead. We analyze here at a high level the communication and matching overheads of the protocol. We will describe these experimentally later. The communication overhead for the metadata is the size of the Bloom filter, which is proportional to the number of keywords stored in it. We have already mentioned that to match document content a small number of keywords suffices (e.g. 50), and for path matching the depth of the path is lower bounded in practice (the author's filesystem has a maximum depth of 22).

The parameters of the Bloom Filter are the size of the filter, m, and the number of hash functions r. Assume we wish a false positive rate of 1 in 100,000 (which should return very few false matches, for

large numbers of files); the optimal value of r is 17, we would use 25 bits for each element on average, so $m = 25 \cdot 50 = 1025b \simeq 130B$.

The query simply lists the r positions of the bits in the Bloom filter, so the expected size is $r \cdot \log m = 170b \simeq 22B$.

The matching overhead is dependent on the number of hashes computed; when query does no t match the metadata, on average $r/2$ hashes will be computed by the server; when the query matches the metadata r hashes will be computed.

Dictionary Keyword Matching

The scheme proposed by Chang et al. [CM05b] is based on creating a dictionary that has one bit for every possible word (as opposed to the words in that specific document). The dictionary is shuffled using a pseudorandom permutation and blinded using pseudorandom functions and a random nonce. The metadata includes the blinded dictionary, along with the random nonce.The query contains the shuffled index of the word plus a "hidden" version of the index.

Let F, G be two pseudorandom functions and E be a pseudorandom permutation. The *Dictionary* scheme is:

`Keygen(`t`)`**:** select $K = \{K_1, K_2\}$ uniformly at random from $\{0,1\}^t \times \{0,1\}^t$.

`EncryptQuery(`K`,`S`)`**:** find index λ of S in the dictionary D. Return $\{index = E_{K_1}(\lambda), F_{K_2}(index)\}$

`EncryptMetadata(`K`,`N`)`**:** let J and I be two bit index strings of size $|D|$, initialised to 0. For all words $w_1, \ldots, w_n$ in N, find λ_i (the index of w_i in the dictionary) and set $I[E_{K_1}(\lambda_i)] = 1$. Select a random nonce rnd. For $i = 1 \ldots |D|$, compute $r_i = F_{K_2}(i)$ and set $J[i] = I[i] \oplus G_{r_i}(rnd)$. Return ($rnd$, J)

`Match(`N_e`,`S_e`)`**:** Let S_e = ($index$, r_{index}). Let N_e = (rnd, J). If $J[index] \oplus G_{r_{index}}(rnd) = 1$ return 1, otherwise, return 0

`Cover(`S_1`,`S_2`)`**:** Return 1 if $S_1 = S_2$, 0 otherwise.

Theorem 3. *Dictionary* is a correct implementation of PPS.

Proof Sketch. Definition 7 provides a security model for PPS regardless of the query, by mandating that the information the server can learn by using the messages received from the user can also be learnt by accessing an oracle. The security model provided by Chang et al. [CM05b] is an instance of our model, where the query function is keyword matching and the oracle is replaced by access to the actual information (i.e., which document contains which keyword). The difference between their model and ours is the treatment for queries (keywords). They assume that all keywords are different (and therefore no information is gained by seeing they are different), while we allow the server to distinguish whether one query covers another query. In the case of keyword matching, two queries cover one another only if they are equal. If we only consider the subset of distinct queries, we can directly use the security proof in Chang et al. [CM05b] to prove security in PPS. The redundant queries do not leak any additional

information about documents, and do not leak more information about queries that cannot be discovered by using the oracle. Therefore, *Dictionary* is secure according to PPS.

Overheads. Compared to *Bloom Filter Keyword*, *Dictionary Keyword* does not generate false positive matches and does not impose any restrictions on the number of words in the document.

This scheme assumes the dictionary is known before the metadata are created, and that it stays constant during the metadata lifetime; if words are added to the dictionary afterwards, all the metadata for all documents must be recreated.

The communication and storage overhead of metadata encrypted by *Dictionary* is equal to the size of the dictionary. The size of the encrypted metadata is 32kB for documents written in the English language [CM05b]. This is very expensive for small documents. The expected size of document content in PPS is quite small (hundreds of bytes usually) favouring the first scheme. *Dictionary* can be used when the size of the string is larger or comparable to 32kB or in cases where the dictionary is smaller.

Matching with *Dictionary* is cheaper, as a single one way function computation is required. Compared to the *Bloom Filter* approach, it is a few times faster on average.

Beyond Single Keyword Queries

The schemes above are secure, yet they allow one to match a single keyword at a time. To match two keywords, there are two straightforward options.

The first is to just encrypt the two keywords separately, submit them to the server, and have the server return the results matching both keywords. This leaks more information than necessary to the server, as the latter knows all documents that match either one of the keywords, not just those that match both.

A second option is to have the user to submit one keyword to the server, download all the matching documents, and match the second keyword locally. This also leaks more information to the server (more documents match) and also wastes bandwidth.

We propose a new solution to this problem, that works if the number of keywords allowed in a search is small (say 2). The basic idea is to create every possible combination of keywords and list documents as having or not having that combination. Single keywords are a special case of keyword pair, where the second keyword is empty.

Is this scheme practical? The average number of keywords in web searches is 2.3, so we believe allowing two keywords should suffice in the vast majority of cases. To estimate communication costs, let us assume we use *Bloom Filter Keyword* as a basis, and that we list only the 50 most important keywords in each document. In the resulting encoding, we would have $50^2 = 2500$ entries in each document, which equates to about 7.5KB with a 1 in 100.000 BF encoding. Whether this is practical depends on the size of the document, and the update frequency.

5.5.3 Numeric Matching

Matching numeric attributes is important as such searches are frequent in filesystem searches, either for searching newer files or files of a certain size. Further, the ability to match numbers can help implement more advanced keyword searches that would be useful in practice, such as ranked searches.

Let $D \subset \mathbb{R}$ be the numeric metadata space. Given a metadata $N \in D$, the query can have two forms: a) inequality tests ($N > l_b$, $N < u_b$) or b) range tests ($l_b < N < u_b$), for $l_b, u_b \in D$. We define two novel PPS schemes for the two cases.

Supporting Inequality Queries

Choose l points, $p_1, \ldots, p_l \in D$ as reference points. We consider the following dictionary: {"$> p_1$", "$> p_2$", ..., "$> p_l$" "$< p_1$", "$< p_2$", ..., "$< p_l$"}. Queries will be approximated with one of these constraints. Each metadata N is considered to be a document containing the words in the dictionary that it matches. These are encrypted using either one of the two *Keyword* schemes we have previously described. The *Inequality* scheme is:

`Keygen(`r`)`**:** K = *Keyword*.`Keygen`(r). Agree on a set of l reference points $p_1, \ldots, p_l \in D$.

`EncryptQuery(`K`,`S`)`**:** Let $S = (type, value)$, where $type$ can be "<" or ">". Find i such that $|value - p_i| = \min_{j=1}^{l} |value - p_j|$. Return *Keyword*.`EncryptQuery`($K, type|p_i$)

`EncryptMetadata(`K`,`N`)`**:** Let $N_w = \{t_i|p_i$, where t_i="$>$" if $N > p_i$ and t_i="$<$" if $N < p_i$, for $i = 1 \ldots l\}$. Return *Keyword*.`EncryptMetadata`(K, N_w)

`Match(`N_e`,`S_e`)`**:** return *Keyword*.`Match`(N_e,S_e)

`Cover(`S_1`,`S_2`)`**:** we check whether the queries are the same by using *Keyword*.`Cover`. Full query covering cannot be checked without additional information in this case. We present an efficient solution in the Implementation section, which leaks some additional information.

Theorem 4. Suppose all queries can be expressed exactly using the mechanisms above. Then, *Inequality* is a correct implementation of PPS.

Proof Sketch. *Inequality* is an instance of *Dictionary* that contains as words "$> p_1$", "$> p_2$", ..., "$> p_l$" "$< p_1$", "$< p_2$", ..., "$< p_l$". Since the approximation is assumed to be perfect and *Dictionary* is secure (Theorem 3), verifying inequality using the dictionary gives as much information as verifying with the oracle. It follows that *Inequality* is also secure.

Note that the assumption that queries are expressed exactly is important. Without this, the server can infer additional information. Here is a simple example: assume the notification space is $0, \ldots, 10$ and the reference points are $0, 5, 10$. Query $S = x > 7$ will be approximated with $S_a = x > 5$. Given encrypted notifications 4 and 6, the server cannot distinguish them in the ideal case (when testing against S, none of them is matched), however it can tell they are different in reality (as S_a will match 6 and not match 4).

Overhead. The overhead of this scheme is due to the size of the dictionary, equal to $2 \cdot l$. There is a direct tradeoff between this overhead and the precision it allows for queries.

If we want perfect queries (0 false positive and negative matches), we set $l = |D|$. This can be expensive in reality (e.g., for 4 byte integers we have $\sim 10^9$ points). We describe an exponentially spaced partitioning scheme that is useful in many practical scenarios. Approximating the 4 byte positive integers with $[1 \ldots 10^9]$, we select as reference points:

$1, 2, 3, \ldots, 10, 20, 30, \ldots, 100, 200, 300, \ldots, 1000, \ldots, 10^8$
$, 2 \cdot 10^8, 3 \cdot 10^8, \ldots, 10^9$. Although the number of reference points is only 100 (the metadata has only 12 bytes), the precision is acceptable if we consider that query sensitivity decreases as metadata values increase.

Supporting Range Queries

To support $l_b < N < u_b$ queries, our initial idea was to have the user create a partitioning $P = \{p_1, \ldots, p_l\}$ of D. The user would encrypt the index of the subset N belongs to by using *Equal*. Queries are mere encrypted versions of the indexes of the subsets in the partition they are interested in (i.e., all $p_i \in P$ such that $p_i \cap (l_b, u_b) \neq \emptyset$). However, sending multiple subsets leaks more information than necessary. Therefore, we would have to approximate the query with a single subset in the partition. As query sizes are not fixed a-priori, we can either grossly overestimate the size of the query, leaking more information to the server and wasting bandwidth, or we can underestimate the size of the query, which also leaks information but does not waste bandwidth.

The initial idea can be refined as follows. Create several partitions of D, $P_1, \ldots, P_m$, with different subset sizes and different starting offsets. Create a dictionary containing as words the index of the partition concatenated with the subset index, for all m partitions. A metadata can be expressed as a document with this dictionary by listing the subsets it is included in. The query is approximated with one of the subsets in these partitions. The *Range* scheme is:

`Keygen(`r`)`**:** Generate K using *Keyword*.`Keygen`. Agree on m partitions of D, $P_1, P_2, \ldots, P_m$, where $P_i = p_{i,1} \cup p_{i,2} \ldots \cup p_{i,l_i}$. Let $p_{i,j} = [a_{i,j}, b_{i,j}]$

`EncryptQuery(`K`,`S`)`**:** Let $S = (l_b, u_b)$. Find the best approximation of S in $P_1, \ldots, P_m$. In particular, find x and y such that $|l_b - a_{x,y}| + |u_b - b_{x,y}| = \min_{i=1}^{m} \min_{j=1}^{l_i}(|l_b - a_{i,j}| + |u_b - b_{i,j}|)$. Return *Keyword*.`EncryptQuery` ("x, y")

`EncryptMetadata(`K`,`N`)`**:** Let N_w = {"x, y"| where $x \in \{1, \ldots, m\}$ and $y \in \{1, \ldots, l_x\}$ such that $N \in p_{x,y}$. Return *Keyword*.`EncryptMetadata`(K, N_w)

`Match(`N_e`,`S_e`)`**:** return *Keyword*.`Match`(N_e,S_e)

`Cover(`S_1`,`S_2`)`**:** we can easily check to see if two queries are the same by using *Keyword*.`Cover`. However, we cannot properly check full covering without additional information. In [RR06] we describe an efficient coverage solution that can be used instead, but leaks more information than necessary

Theorem 5. Suppose all queries can be expressed exactly (i.e., without generating false positives or negatives) using the above algorithm. Then, *Range* is a correct implementation of PPS.

Proof Sketch. Same reasoning applies as for Inequality.

The scheme creates an explicit tradeoff between the size of the queries and matching time on one hand, and the number of false positives and the security attained (i.e., information leaked due to imprecise

queries), on the other. A partitioning scheme with zero false matches for any range query has $|D|^2$ points, being quite expensive. A better scheme can be obtained if we focus on query sizes likely to be used in practice.

In general, given a desired cost, choosing the proper partitioning is application specific and should take into consideration the distributions of queries and metadata. An algorithm that determines the optimal partitioning strategy for a specified cost is presented by Hore et al. [HMT04] and could be used for this task.

5.5.4 Supporting Ranked Queries

We are now ready to describe our construction for ranked queries. In traditional information retrieval, the document score is computed using a scalar product between the query and the document vector representations. It is difficult to implement this exact functionality with PPS, but we can approximate it quite well if we have few keywords in the query.

First, assume there is a single keyword in each query and we wish to match only those documents where the keyword is of utmost importance, say in the first five features of the document. To allow such matching, we create the following partitioning of the feature space: first, first five, first ten, and first 25. If a keyword satisfies the query (i.e. it is first), an encryption of "first—keyword" will be added to the document. All in all, we add 41 new keywords to each document, which increases the size from 130B to 250B.

If we allow dual keywords in each query, and we wish to maintain the same allowed ranking (first 1%, first 5%, etc.) we roughly double the size of the metadata to about 15KB.

5.5.5 Supporting Generic Queries

Supporting arbitrary functions as queries is not a goal in itself, as the maximum achievable security is not satisfactory: Only $O(|N|)$ carefully chosen queries are enough to distinguish every metadata. This, combined with the knowledge of a plaintext-ciphertext pair, completely breaks the metadata encryption scheme. However, it is interesting to discuss approaches for generic query functions as a possible starting point to support other query functions of practical interest.

There is a tradeoff between the amount of information leaked to the servers and the communication overhead. Therefore, to support generic queries we can trade confidentiality for communication efficiency.

At one end of the solution space, the minimum amount of information is revealed and communication size is very expensive. Consider an enumeration of all functions from $D \rightarrow \{0, 1\}$. The dictionary will contain the indexes of all these functions. We use *Dictionary* to encode arbitrary queries by encrypting the proper index. Metadata will include as words all the indexes of functions that accept them. This scheme is secure for all possible queries as it does not leak more information than what is needed. The communication size is huge: Every metadata has $2^{|D|}$ bits.

At the other end of the solution space, we have examined and implemented a protocol based on Yao's garbled circuit construction to support generic queries, expressed as boolean circuits [Yao86]. The size of the communication is small (query size is directly proportional to the number of gates in

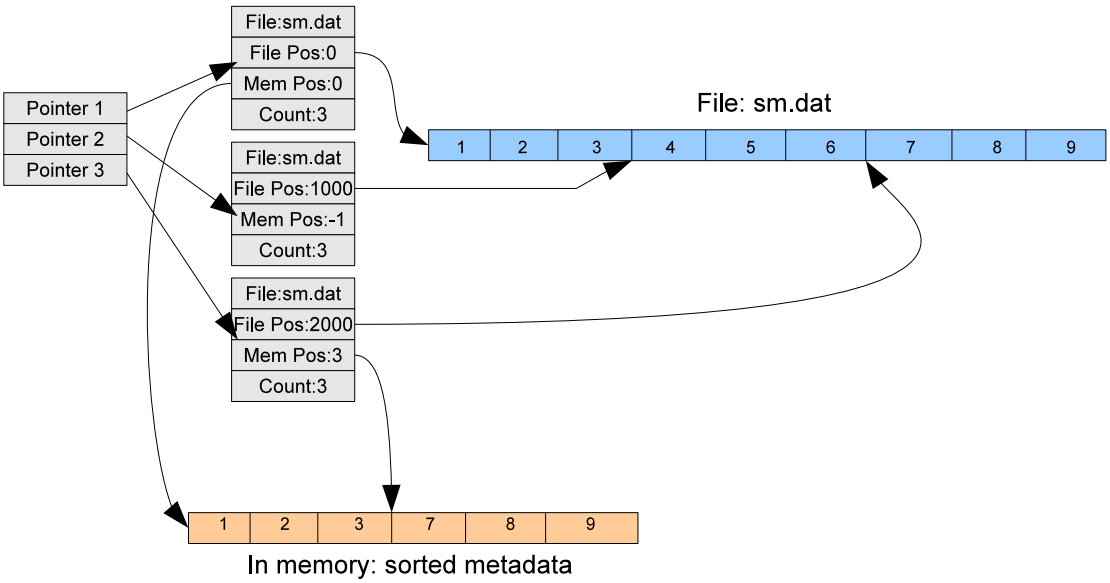


Figure 5.2: Data Structures Used by PPS

the circuit, while metadata size is the same as the plaintext version). However, this scheme allows the server to distinguish every bit of the metadata, and therefore a single plaintext-ciphertext pair is needed to completely break metadata (without needing |N| "good" queries as a basis).

5.6 Implementation

We implemented all the algorithms we presented in Java 1.5. We chose Java mostly for ease of development and debugging. The only concern we had was for performance, but techniques such as HotSpot JIT compilation make Java reasonably fast.

We used the SHA-1 cryptographic hash function [oST95] throughout our implementation as a pseudorandom function. We used 128-bit AES [DR02] for the symmetric encryption scheme and as a pseudorandom permutation.

5.6.1 Overview

The server stores for each user all the metadata the user has registered. Multiple users will be serviced by the same server as multiplexing is needed to make PPS economically viable.

The user provides a random identifier for each metadata. The server code loads the metadata from disk into memory in the increasing order of the identifier, performs the matching in memory, and returns the results. A user's metadata is cached as long as memory is available. When a user submits a query, the query is served from memory if the user's metadata is in memory. Otherwise, if memory is full a user's metadata will be deleted from memory. The cache policy is least recently used (LRU).

Caching improves performance when a user emits a burst of queries in a short period of time. A server will need to service a large number of users, a small subset of which is active at any point in time, so we expect that in the common case the user's data is not in memory when the query arrives.

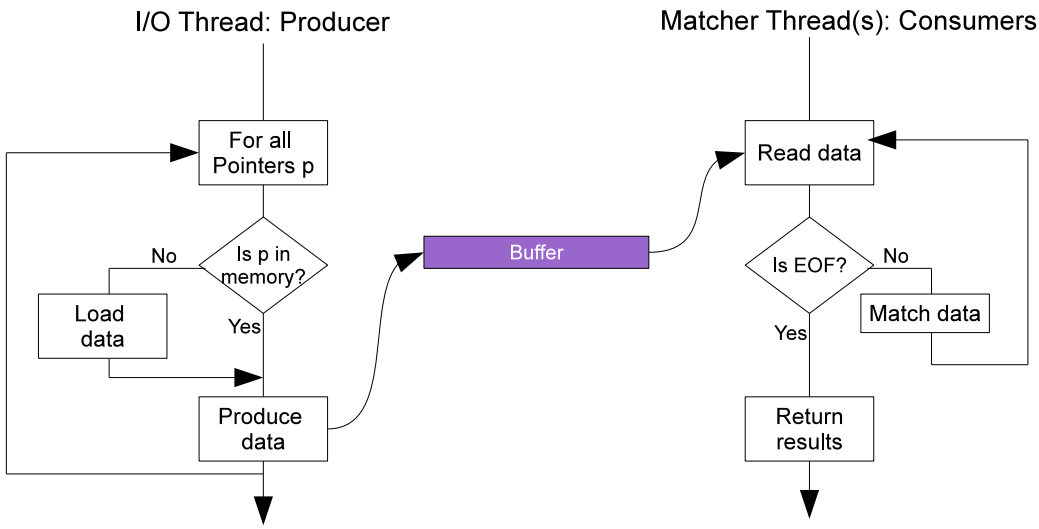


Figure 5.3: Running a Query with PPS: System Architecture

5.6.2 Managing Metadata

To manage metadata, we created a data structure that allows partial loading and quick access to entries. Partial loading is used when a single query is split across many servers, and each server only matches a subset of their local data (i.e. when increasing p_Q with ROAR). The data structure is based on an array of user metadata sorted by id. Disk storage uses the same structure, storing metadata sequentially in one or a few files on disk. Besides the array structure, we maintain an array of "pointers" to these basic lists, to allow fast and partial access. The data structures are presented in Figure 5.2.

When a user sends a query the server will create an in-memory sorted metadata list (if one does not exist already), loading the pointer entries from a small file on disk. Initially, there is no metadata in memory.

The range requested by the query is used to select ranges of metadata to be loaded, and the server begins loading data from disk (file "sm.dat" in our example) using information from the relevant pointers. To load data from disk it uses memory mapped I/O (this is faster than traditional I/O for large files). An I/O thread will sequentially read from disk into the in-memory list the data corresponding to each pointer entry. In our example, data is read for pointer 0 and placed at position 0. Then data from pointer 2 is read and placed at position 3 in memory.

5.6.3 Running Queries

To run the query, metadata are loaded (if necessary) and then matched against the encrypted query. There are several bottlenecks that could appear: loading from disk could be slow, or matching could be slow. To decouple these two, we create two threads: one that reads the data from disk or memory and feeds it to another thread that matches the metadata against the query.

The architecture of the system is presented in Figure 5.3. We use a fixed-size buffer to synchronise the two threads, using the producer-consumer paradigm. The buffer hides I/O latency in the case when CPU is the bottleneck, even in a single processor system. To avoid excessive use of synchronisation, the

I/O thread produces batches of metadata at once, and the consumer only announces the sleeping producer when enough space is available for an entire batch. When the I/O is the bottleneck, the setup adds very little overhead compared to sequential match and load.

Dealing with multi-core servers servers is easy: the code simply creates one matching thread per physical core, and the buffer now has a single producer and multiple consumers. The server supports inter-user query parallelism, but serialises queries from the same user (thus achieving fair sharing).

5.6.4 Metadata Encoding

Each user file has three types of metadata. File size, last modification date and keywords (both filename and, where applicable, the most important keywords in the file contents).

The straightforward way to encode this data would be to encode each attribute separately, and allow predicates to select one of the attributes. This leaks information, as the server knows how many times each attribute is queried, and can infer the attribute type.

The better solution is to embed all attributes into a single visible metadata. This is possible because all practical queries use keyword filtering as a base. We use the same keyword matching algorithm for all attributes, and create a dictionary that is a superset of all the per-attribute dictionaries. For instance, if the keyword dictionary is $\{distributed, systems\}$ and the file size dictionary is $\{1, 2\}$ we can create a dictionary $\{kw = distributed, kw = systems, size = 1, size = 2\}$ that encompasses both. To match keywords, the user will create a query by prepending "kw=" to its desired search keyword. In this way, we can stack up all the attributes in a single dictionary with size equal to the sum of the individual dictionary sizes.

There are no associated space overheads. In the *Dictionary* scheme total metadata size will be the same as if we had encrypted each metadata individually, and the same applies to the *Bloom Filter Keyword*.

5.6.5 Multi-Predicate Queries

In our implementation a query can contain multiple predicates and a binary function (*and*, *or*) to aggregate the results. We have mentioned earlier that while multi-predicate queries leak more information than needed to the server, supporting all possible multi-predicate queries securely has prohibitive costs. Hence, we allow multi-predicate queries for practical purposes. It is the user's choice whether they wish to use multiple predicate queries or single predicate, perfectly secure queries.

The matching algorithm initially runs all the predicates in the query regardless of the binary function, counting the number of matches for each predicate (we call this predicate "selectivity"). After a small number of samples, it sorts the predicates according to their selectivity, and starts to match predicates selectively. If the binary function is "*and*", it will apply the most selective predicates first; if the binary function is "*or*", it will apply the least selective predicates first.

As the matches are randomly scattered through the metadata, matching a few metadata is provably enough to get a very good estimate of each predicate's selectivity.

Here is a succinct explanation. Let the predicate's real selectivity be s'; we match n metadata chosen randomly to find an estimate s of the selectivity. The number of matching metadata in n samples

is a random variable that has a binomial distribution with mean ns and variance $ns(1-s)$. Using Chebyshev's inequality and $\sim 89\%$ confidence we have:

$$Pr(|ns' - ns| > 3\sqrt{ns(1-s)}) < 1/9 \tag{5.1}$$

Dividing by n and upper-bounding $s(1-s)$ by 1/4, we get $|s' - s| \leq q\sqrt{\frac{s(1-s)}{n}} \leq \frac{3}{2\sqrt{n}}$. To get an accuracy of 0.1, it suffices to set $n = 225$. This is the number of samples we use in our implementation.

5.7 Evaluation

Privacy Preserving Search must be fast to provide a good user experience. In this section we explore the performance properties of PPS for typical numbers of files likely to appear in practice. Our experimental setup uses file information from the author's home directory to generate metadata as described before. The metadata is stored and experiments are run on a Dell PowerEdge 1950 server with 2GB of main memory and two dual core Intel Xeon 5150 processors running at 2.66GHZ. We used the Linux operating system, with kernel version 2.6.28. We experimented with queries matching as little as 10000 files up to a few million, covering a wide enough range to gain a good understanding of the performance limitations of PPS.

Our basic experiments use a set of 1 million metadata which are repeatedly queried by a server in the same LAN using two random keywords, such that the number of matched metadata is always 0 (this is to avoid measuring the network cost of transmitting the data back to the client).

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[2]. PPS_LM (low memory) forces a run of the garbage collector immediately after finishing a query. This has 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. Unless stated explicitly, we run PPS_LM by default in our experiments.

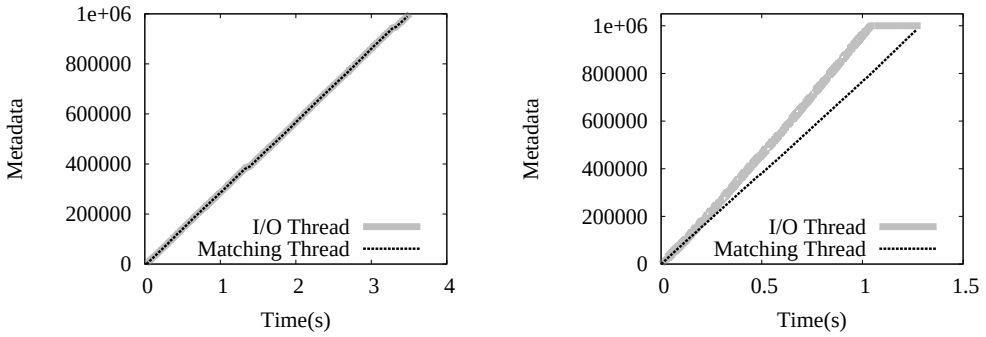
Basic Performance. We wish to understand the scaling bottlenecks of PPS. We first ran the query with cold disk caches and a single matching thread, and found that mean end-to-end query delay is 3.9s, with all values within 0.5s of the mean. To understand the bottleneck, we instrumented the producer-consumer buffer to output a line whenever a multiple of 1000 metadata are produced or consumed. We plot the results in Figure 5.4(a) for one of the queries.

In the plot the I/O and matcher thread lines perfectly overlay, indicating that the producer—the I/O thread—is the bottleneck. To verify this assumption, we ran a simple tool that just reads the whole 230MB metadata file [3] and found it took around 3.5s to complete. The remaining 0.3s are accounted for by the list append operations the I/O thread performs.

The metadata file was written sequentially on disk, and was also read sequentially. We wondered

[2]Memory allocation is however faster in Java than C++, because the free space is contiguous, as the heap is compacted on each collection

[3]The command printed the number of characters in the file: *cat file | wc -c*



(a) Query reading data from HDD

(b) Query reading data from OS buffer cache

Figure 5.4: Execution traces for queries searching 1 million metadata

whether the ext2 filesystem was causing the performance problems. Further experiments show this is not the case: the maximum achievable transfer speed is given by the sequential raw hard disk transfer speed of 85MB/s, which we measured with "hdparm". The transfer speed in the experiment above is around 66MB/s, 75% of the optimal. Even if we achieved the maximum speed, and with no other overheads the query would still take 3s to complete. 3s per query is too slow to be acceptable for regular users.

Warm OS Buffer Caches. We expect many users to be multiplexed on the same server, so it is very likely that user queries will be interleaved and caches nearly always cold. Such queries will be disk-bound and performance will suffer. However, it is equally interesting to explore what happens when there is query locality, and caches are warm. Modern operating systems maintain a "buffer cache" where recently read data is cached. Linux in particular is quite aggressive, using all available memory for the buffer cache.

If the same user runs a burst of queries, only the first query data will read data from disk. With high probability subsequent queries will access data in the OS's buffer cache. We repeated the experiments above, but with warm OS buffer caches.

With a single matching thread query delays are around 1.4s, much faster than 3.9s with cold caches. A look at the output of the producer-consumer buffer shows that in this case the bottleneck is the matching thread, which lags behind the I/O thread (see figure 5.4(b).

CPU-Bound Queries. We profiled the execution to understand the CPU overheads. A query running all the operations with in-memory metadata and without performing the matches takes 0.3s: this overhead is mostly due to adding/removing items from lists in Java. The remaining 1.1s are due to matching the metadata. Most of the time is spent in calling the SHA-1 function which was called approximately 2.5 times per metadata in our experiments. Typical SHA-1 implementations take 8 processor cycles per byte to execute [KKG+10]. Our processor's speed is 2.6Ghz and the keyword metadata is 140B, so the processor can run at most 2.32 million SHA-1 function applications per second. For a query against 1 million metadata roughly 2.5 million SHA-1 function applications are needed, taking around 1.1s. This matches our profiled execution time.

The number of SHA-1 applications per metadata (on average 2.5) is upper bounded by the number

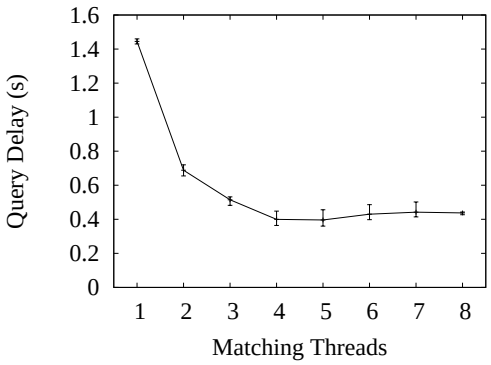


Figure 5.5: Query delays with in-memory data and different number of matching threads

of hash functions used by the keyword bloom filter (17), which instead depends on the total number of keywords in the bloom filter and on the probability of getting false positives. We used 17 hash functions to get a false positive match probability of 1 in 100.000.

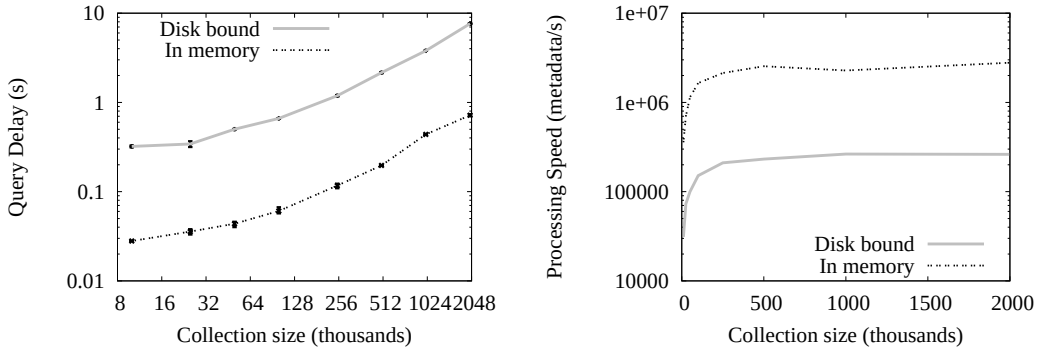
The CPU overheads grow linearly with the number of hash functions applied. When a query matches some metadata all hash functions are applied to verify the match and CPU costs are highest. If a query matches all metadata, query delays increase sevenfold, as all 17 hash functions are applied for every metadata being searched.

However, these high costs do not appear in practice for two reasons. First, if a query does match everything, the query will be stopped early and the first few hundred matching results will be returned. The second case is when a multi-keyword query matches few entries, but some of its keywords match all (or most) entries (e.g. when searching for “the doors” “the” will match nearly all documents). In this case the server will order predicates in increasing order of selectivity, and query delays will be reduced. This is why we ran our tests using queries that did not match any metadata.

To speed up execution of CPU-bound queries, we can increase the number of matching threads; each of these will be scheduled onto different cores, so we expect significant speedup. Surprisingly we found that query delays quickly reach a plateau at 1.1s when increasing the number of threads to two; further increasing the thread count yields no improvements. On closer examination, we see that with two or more threads, the I/O thread is the bottleneck again. In this case the overheads come from system calls, parsing the data, allocating memory, etc. These were “masked” by the I/O delays when the system was disk-bound.

In-memory cache. We can bypass all these overheads with an in-memory metadata cache. The memory usage is similar to the buffer-cache and there are no costs when running new queries. The cache has an upper bound of metadata items to be stored and uses the LRU replacement strategy.

We enable the cache and plot the query delays in Figure 5.5 as a function of the number of matching threads. Up to four threads, each thread is scheduled onto a different core and the speedup is almost linear. With four threads, one query only takes 400ms on average. Increasing past 4 threads only decreases performance due to increased locking and scheduling costs.



(a) Query delays scaling

(b) Processing speed scaling

Figure 5.6: PPS performance scaling with file collection size on a Dell 1950

5.7.1 Dynamic predicate ordering

To evaluate the effectiveness of dynamic predicate ordering, we ran a simple experiment searching for "the xyz" that returns zero matches. With in-memory data, one matching thread and predicate ordering enabled the first few hundred objects are matched against both keywords and after that predicates are sorted such that the more selective "xyz" is matched first. Beyond this point, all metadata are only matched against "xyz", and queries take on average 1.25s.

Next we turned dynamic ordering off and ran the query "xyz the": the server applies the predicates in the user-provided order, and the mean query delay is also 1.25s. This shows that the overhead of matching the first 225 metadata against both keywords is negligible. Finally, we ran the original "the xyz" query. Query delay in this case grows to a surprising 10s. Of these, 8.75s are due to matching "the" and 1.25 to "xyz". The increased costs of matching "the" are due to the many more SHA-1 applications (17 vs 2-3 per object).

Dynamic predicate ordering is very simple and cheap. Its most important benefit is that it allows query delays to be independent of the query terms and count for queries that have "wildcard" keywords. Predictable performance makes it easier to provide good and predictable search response times to users.

5.7.2 Query delays with varying numbers of metadata

The results we have presented provide an accurate image of overheads when running queries against 1 million metadata. How do these results scale up and down with larger or smaller collections of files? We ran experiments with collections of files containing as few as ten thousand files (20MB on disk) up to two million files (500MB on disk).

We present the query delays in Figure 5.6(a) using a log-log scale. As we increase the number of items of metadata that must be searched, query delay increases. Query delay scales linearly with the number of metadata objects when there are large numbers of objects to be matched for both disk-bound and CPU-bound processing. The in-memory experiments were run with 4 matching threads, to get the best performance.

In Figure 5.6(b) we plot the server processing speed for the same experiments. When the number

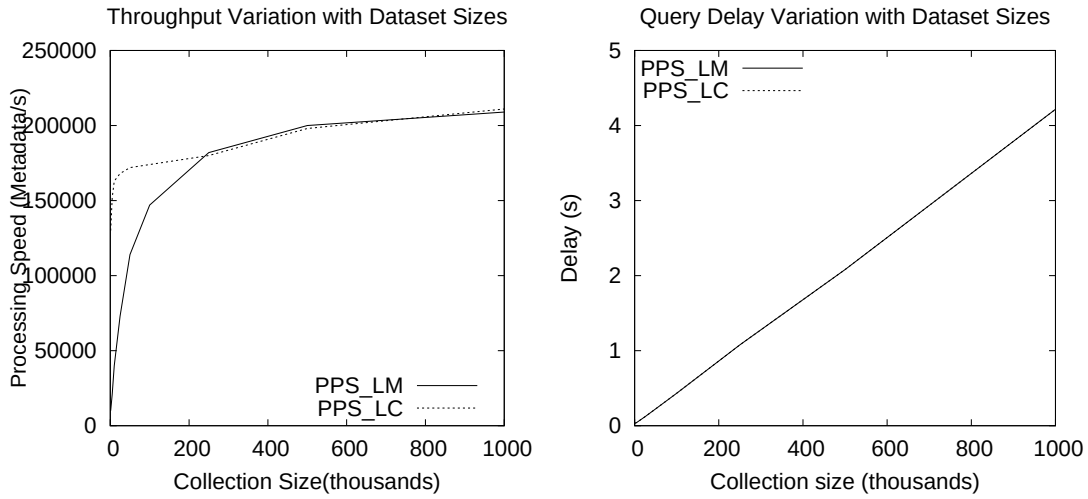


Figure 5.7: PPS performance scaling with file collection size on a Sun X4100

of objects is smaller, the server processing speed is lower. This is because the fixed costs associated with a query cease to be negligible. These include network related delays due to connection setup, data transmission, parsing and serialisation of the query and results, and so on. Other host related fixed costs include starting new search thread(s), providing end-of-query thread synchronization, allocating and releasing memory, etc. For disk-bound processing the fixed costs also include the disk seek times (on the order of 10ms per seek); for small files these are not amortised over long sequential reads. These fixed costs start to be amortised by the time the server is searching about 100,000 files, and at 250,000 files the throughput of the server levels off for both curves.

As a high level point, disk-bound query delays exceed 1s at 250,000 metadata, and increase linearly past that. We need to parallelise such searches to provide a good user experience; we will use ROAR for that purpose. Even in-memory processing quickly reaches single-server scaling limits. On a four core server delay is 700ms for 2 million metadata. As collections of files become bigger, running searches on a single server will quickly result in bad performance. Either it will take too long to match the data or the metadata will grow bigger than the server's memory [4] and the query will become disk-bound.

Distributing computation across many machines is required to make PPS scale. Interestingly, both delay curves have similar shapes, albeit at different performance levels. This implies it should be possible to apply the same parallelisation techniques regardless of the bottleneck.

Different Hardware. So far we have examined how PPS_LM performs on a single Dell 1950 machine, finding that the disk is the performance bottleneck. We re-ran the same experiments on different hardware too, and found that the behaviour is similar for the more powerful Dell 2950 machines. However, for the slower Dell 1850 and Sun X4100 machines the CPU is always the bottleneck, even when the metadata is loaded from disk. These experiments analyzed both versions of PPS (PPS_LM and PPS_LC). The main issue is query delay as shown in Fig. 5.7. As before, once the fixed costs are satisfied, query delay increases with the number of metadata objects to be searched.

[4] This limitation is true for Google's web search [Dea]

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 right-hand graph in Fig. 5.7. The drop is steeper for the low memory version.

5.8 Related Work

The security work in this chapter has been undertaken by the authors in 2005-2006 and was published as [RR06] in the context of achieving privacy in content-based publish/subscribe systems. At that time, to the best of our knowledge, ours was the first complete and secure solution for content-based publish/subscribe that had been presented in the literature; further it was the only secure construction for numeric queries.

The biggest assumption of the initial work paper was that a key was shared by publishers and subscribers. In PPS publishers and subscribers are replaced by the user, and this assumption is no longer needed. From the security point of view applying the same techniques to Privacy Preserving Search is not only possible but straightforward.

Secure Function Evaluation. Research in cryptography has produced many important results in the broad area of secure function evaluation [FKN94, IK97, CGKS95]. Although several protocols in this space resemble and appear applicable to the PPS problem, none is of practical importance for PPS. First, the protocols have been designed for single invocations and are vulnerable when the same key is used to send multiple queries. For instance, the information-theoretically secure protocol described by Ishai [IK97] can be broken easily when used for multiple queries, while the semantically secure protocol described by Feige [FKN94] becomes as secure as the one time pad in the same context. In theory, we can use such single message protocols in the context of PPS, but with tremendous overhead: For every query, the user would re-encrypt all their data and store it online. Further, even the cheapest instances of these protocols have high costs for single invocations.

Privacy Preserving Keyword Searches. Motivated by public file servers and email servers, a more practical approach was taken by the security community to solve the problem of searching encrypted files using keywords.

The pioneering work in this direction is due to Song et al. [SWP00], who propose a scheme that encrypts each word in the document in a way that allows a user to search using an encrypted keyword. To test whether a given keyword is in an encrypted file, a sequential scan of the file is needed; this approach does not scale well for large documents.

Schemes were proposed by Goh [Goh03b] and Chang et al. [CM05a] that use indexes to address this issue and propose stronger security models. For practical reasons, we used the first scheme for keyword search and the second as a basis for supporting range matches. Our work employs a security model that is similar to the one from Chang et al. [CM05a], extended to deal with arbitrary subscriptions and to allow subscription covering (that was implicit in the initial model). Our mechanisms can be used to provide privacy preserving range matches for numeric values.

Curtmola et al. proposed an improved security model and more efficient constructions for privacy

preserving keyword search concurrently to our work [CGKO06]. The authors observe that previous definitions of security in the area such as [SWP00, Goh03b, CM05a] do not cover the case of multiple queries, and create a security model that includes search history to address this shortcoming. Our security model, initially proposed in the context of content-based publish/subscribe also incorporates query history.

The main idea in [CGKO06] is to create an inverted index-like structure which is encrypted and stored on the server. This data structure consists of a lookup table of terms, and each term has associated a linked list containing the matching document identifiers; these are then blinded and scrambled to stop the server from gaining information about documents. This index can be searched efficiently, in constant time, regardless of the number of documents. This differs from the schemes of Goh and Chang where CPU search time is linear in the size of the document collection. We have seen, however, that search for PPS is likely to be disk-bound, thus CPU speed does not matter that much.

Compared to the naive solution of just using the encrypted inverted index and downloading it before queries, Curtmola's scheme has the advantage that it does not need to download the index. However, the index must be kept updated, and potentially the whole index needs to be changed when documents are changed, added to or removed from the collection; the resulting bandwidth overheads would be prohibitively high. This overhead could be reduced by trading-off index accuracy against update bandwidth, though. Further study is needed to decide which PPS constructions are better in practice.

5.9 Conclusions

We have shown that it is possible to perform encrypted searches against encrypted data on untrusted servers. We have used existing keyword matching constructions as a basis to support numeric and more general matching.

PPS is less costly for mobile users than the straightforward approach of downloading and decrypting an encrypted index for searching, and will be preferable in the near future where more users access their data on the move.

However, Privacy Preserving Search takes too long even when modest numbers of files are searched: with 250,000 files it takes in excess of 1s to get the results, mainly because disk access is the bottleneck. To make Privacy Preserving Search practical we need to parallelise it across many servers. In Chapter 7, we will show how to do just that with ROAR.

Chapter 6

Analytical Evaluation

We wish to understand the fundamental properties of PTN, SW and ROAR, including query delays, bandwidth consumption, load balancing, fault-tolerance and the ability to change p dynamically. A first step in our exploration is this chapter, where we use a mix of analytical modelling and simulation to distill the properties that determine algorithm behaviour.

We restrict our analysis to the basic Distributed Rendezvous operations (storing objects, running queries, and changing p), and ignore other costs related to practical implementations. The practical costs include bandwidth consumed for control traffic, imperfect load balancing, and so forth. We include these in our experimental analysis (see Chapter 7).

A central point of the analysis is the comparison between query delays obtained by the algorithms when running on a heterogeneous server pool. To guide the analysis, we begin by characterising lower bounds for query delay. Our most important finding is that PTN and ROAR have relatively similar delay values for configurations likely to arise in practice. They also have similar availability for data center-like deployments. ROAR provides more flexibility in adapting the ratio of p and r, and in controlling query delay.

6.1 Query Delay

We want to evaluate query delays given n servers, their partitioning level p, their processing power, and some query arrival rate. To do so, we first provide a model of query delay at a single server.

Definition 8 (Computation Model)**.** Each server has a fixed processing speed cpu, expressed as the number of data objects per second it can match against a query. This assumes that the server takes the same amount of time to match *any* query against a constant size dataset. That is, the server has constant service time.

The query is initiated by the front-end server which splits the query to enough other servers that together can complete the query. The time to initiate a query on another machine and to receive the results is entirely dominated by the round trip time rtt between the two servers plus the local query processing time. If the front-end (server 0) needs to run a query against d data objects on server i, the time required is: $t = rtt_{0,i} + d/cpu_i$.

For simplicity, we assume there are zero or very few returned results, so that bandwidth for returning

results does not impact query execution time. This is true when queries are executed, as few results are returned regardless of the total number of matches (for instance, Google returns 10 results at a time), and is also true in data centers where bandwidth abounds.

Objects are randomly load balanced across servers, which means the number of matches on each server is roughly the same. Therefore, when all servers have homogeneous bandwidth it does not really matter how long it takes to return results from the point of view of the scheduling algorithm.

The definition assumes there is no setup overhead associated with starting a query, and that there are no OS overheads for parsing query requests and sending query replies. We show experimentally that this model is accurate if the query is large enough, and that it breaks down when queries are very small; in the latter case, the setup overheads begin to dominate.

In reality, processing speed varies even for the same machine over time due to OS background activities and concurrent applications. We ignore such effects for now. We make no distinction between CPU and I/O bound query processing, as the linearity factor still holds. Memory-bound query processing is trickier to model, so we assume for simplicity that all the data is either entirely in memory or on disk. We experimentally show that this model is accurate for our target application.

Simulator. We implemented the algorithms and estimates of the optimal delays in a simple numerical simulation. The front-end server has estimates of server processing speeds and maintains for each server a list of tasks assigned and still running.

Queries arrive at discrete times according to a Poisson process with a configurable mean. The scheduler splits each query into exactly p parts and chooses the p servers that would finish first, according to the algorithms described in Section 4.8.1. To estimate query finish time at server i, the front-end assumes the new task will start as soon as server i finishes its last assigned task.[1] The front-end assumes network delays are negligible.

For every query, we log its arrival time and its completion time. We run many queries (a few thousand) to ensure we capture long-term averages. As query arrivals are open-loop, there is a danger that we overload the system with the query load. We test for exploding server task queues by fitting a straight line to the $delay(time)$ function (which gives the delay of a query as a function of its arrival time). If the slope of the fitted line is greater than 0.1 (i.e. query delays are constantly increasing with time), we consider the queue to be exploding and set the measured delay to be infinite; otherwise, we set the delay to the mean of the query delays.

6.1.1 Bounding Optimal Query Delay

We want to find optimal query delay, defined as the average delay of queries run by the system. We mainly wish to understand how the algorithms use servers with heterogeneous computing capacity to improve query delay and increase throughput. We do not examine the impact of network delays on query delays as these are second order effects (only one to a few milliseconds) in data center deployments, our main focus.

[1] This simple model assumes serial execution. It is appropriate for a single-core machine as the scheduler is perfect and can keep it fully occupied; in practice a scheduler will assign a few overlapping tasks to any single core server to ensure good utilisation.

From the computation model it follows that, to optimize query delay, it may be sensible to send the query to more than p servers. For instance, if servers are idle, splitting a CPU-bound query to n servers is faster that splitting it to p servers (although this would increase overheads).

If each query is sent to all servers (i.e. $p_q = n$) a collection of servers will achieve minimum delay if they act as a single server with processing power equal to the sum of each server's processing power. If we assume Poisson query arrivals, the system acts as an $M/D/1$ queue where the service time $D = \frac{1}{\sum_{i=1}^{n} cpu_i}$.

The optimal operating point for distributed algorithms is $p_q = p$, as costs are minimal at this point. If $p_q = p < n$, it is trickier to grasp what the optimal query delay is. If there is very little load, the system is optimal if it can run the query on the most powerful p servers.

As load increases, it is not sufficient to consider only the most powerful servers, as these may become overloaded. When load nears 100%, the system is optimal if it approximates an $M/D/r$ system, with service time $D = \frac{r}{\sum_{i=1}^{n} cpu_i}$.

A heuristic approximation of the optimal in the general case is the following algorithm: sort servers according to their descending CPU power, and assign the first p servers to the first cluster, next p servers to the next cluster, and so on; we will create $n/p + 1$ clusters. When a query arrives, run it on all servers in a chosen cluster, assigning more work to servers proportionally to their CPU speed. The chosen cluster is the one that is estimated to finish the query first.

6.1.2 Query Delay Comparison when $p_q = p$

We ran queries with SW, PTN, ROAR and the theoretical lower bound in a 1000-node network, with 30% query load and server speeds uniformly chosen in the range $0.25Ghz, 2Ghz$.

We run each experiment in two phases, corresponding to different control loops in the algorithms. The first phase is network-setup where the system uses estimates of server processing speeds to setup the data on the servers; in ROAR the speeds are used to compute server range, while PTN uses server speeds to balance compute power across clusters. In the second phase, queries are run. The front-end updates and uses server speed estimates to do query placement.

In practice, the network setup phase runs infrequently, possibly with periodic input from the query execution phase. Thus, the estimates of the phase are in practice much less accurate than those of the query execution phase. However, the network allocation that results in the network setup phase influences the choices available to the front-end in the second phase.

Our simulations model this imperfect knowledge using three different scenarios. First, we assume server speeds never change, such that network allocation in the first phase remains "perfect" throughout the experiment. This gives an upper bound for performance, but is very difficult to achieve in practice due to load the system cannot control (e.g, other virtual machines running on the same box, periodic background OS tasks, or even memory cache self-interference from the query application).

The other extreme scenario is where estimates are useless because the background load changes very frequently. In this case the network is setup assuming all servers have equal performance. This scenario gives a lower bound for query performance.

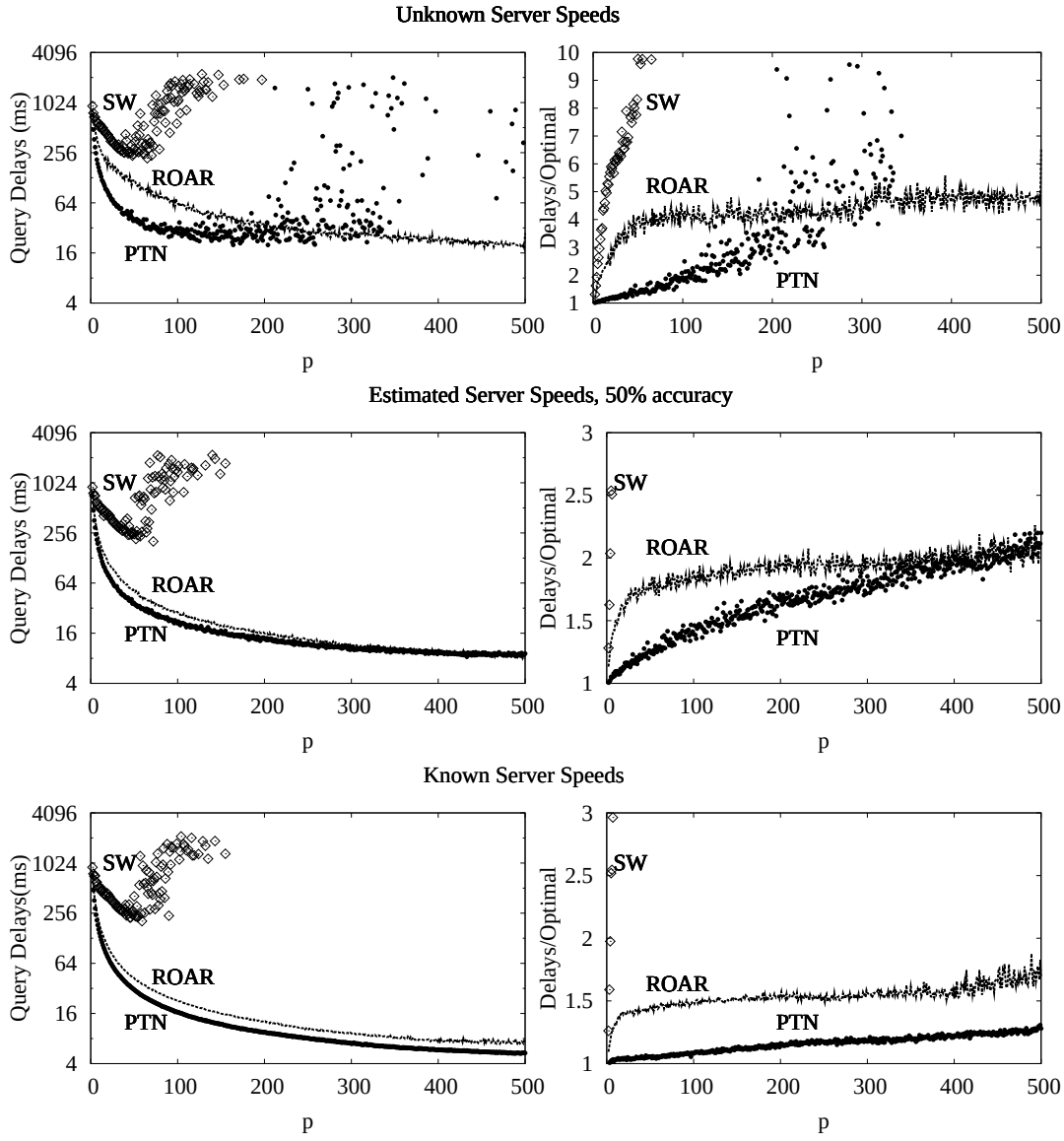


Figure 6.1: Basic Delay Comparison for SW, PTN and ROAR

To understand how things evolve between perfect and zero knowledge, in our third scenario we introduce error into the first phase server speed estimates. In these experiments, if real server speed is x objects/s the system will use an estimate chosen uniformly and randomly in $\frac{x}{2}, \frac{3x}{2}$.

For the three scenarios we first plot the absolute query delays as a function of p in the top part of Figure 6.1. In the bottom part we plot these relative to the optimal described in the previous section.

In the "perfect" case, PTN and ROAR have qualitatively similar performance and are close to the optimal. PTN is on average 15% slower than the optimal. As we have pointed out in Chapter 3, the optimal solution is impractical and requires substantial server movement to maintain the optimal delays. PTN provides good performance with a much simpler structure.

ROAR is 53% slower than the optimal on average, and 33% worse than PTN on average; this a fundamental limitation inherited from SW, where freedom in query placement is limited. Adding more

parameter	Range	Default Value
n	50 to 1000	100
p	1 to $n/2$	-
Speed estimation error	0 to 100	0, 50, Infinity
cpu	uniform, 1x to 128x variation	8x variation
$load$	10% to 99%	30%

Table 6.1: Simulation Parameters

rings does improve latency, but also reduces the flexibility benefits of ROAR.

SW has significantly worse delays, up to ten times the optimal lower bound for small values of p. Further, for large values of p SW does not cope with the load. When p grows the number of sub-queries per node increases, and less powerful nodes will be forced to run more sub-queries. Some of these nodes will become overloaded, and build an infinite queue of queries to service. As SW does not take server speeds into account when creating the network, its performance is identical for all three scenarios.

Moving to the worst case where server speed estimates are unknown, we get an entirely different outcome for PTN. PTN does better than ROAR for small values of p, but becomes increasingly overloaded when p nears 250. At this point, some clusters will contain one less server than the other ones, as p does not divide n. When p is in the 250-333 range, some clusters will have 3 servers, and some 4 servers. The servers in the smaller clusters service 33% more load. When $p > 333$, the imbalance becomes 50%. This effect is compounded with random allocations of servers in clusters: when, by chance a small cluster has only slow servers, they just can't cope with the load.

In comparison, ROAR delays increase to four times the optimal, but is more robust than PTN: increasing p always decreases delay with ROAR. Like SW, ROAR evenly balances query load across all servers, regardless of p, so it avoids the first problem PTN faces. Like PTN, ROAR offers better choice between existing servers, so it always performs strictly better than SW. At this load level, ROAR sees no performance degradation as p increases.

The middle plots in Figure 6.1 show algorithm performance when server speed estimates are inacurrate. Both PTN and ROAR performance are within a factor of two of optimal. PTN still outperforms ROAR, but the gap is smaller (only 15%).

We have examined the impact of server estimates on query delay in greater detail, with results presented in the next sections. Overall, ROAR does better with inaccurate information, for reasons explained above. We expect ROAR and PTN performance to be comparable in practice, where server speeds cannot be accurately predicted.

Parameter Exploration. To gain a deeper understanding of the performance differences between the algorithms we vary n, p, $load$, server speed estimation accuracy, and CPU (the distribution of processing capacity of the servers). We run the same experiments as above. The parameter space is quite large, so a complete exploration is very difficult. To make the analysis tractable, we choose default values for each parameter (see Table 6.1), and vary a small number of free parameters.

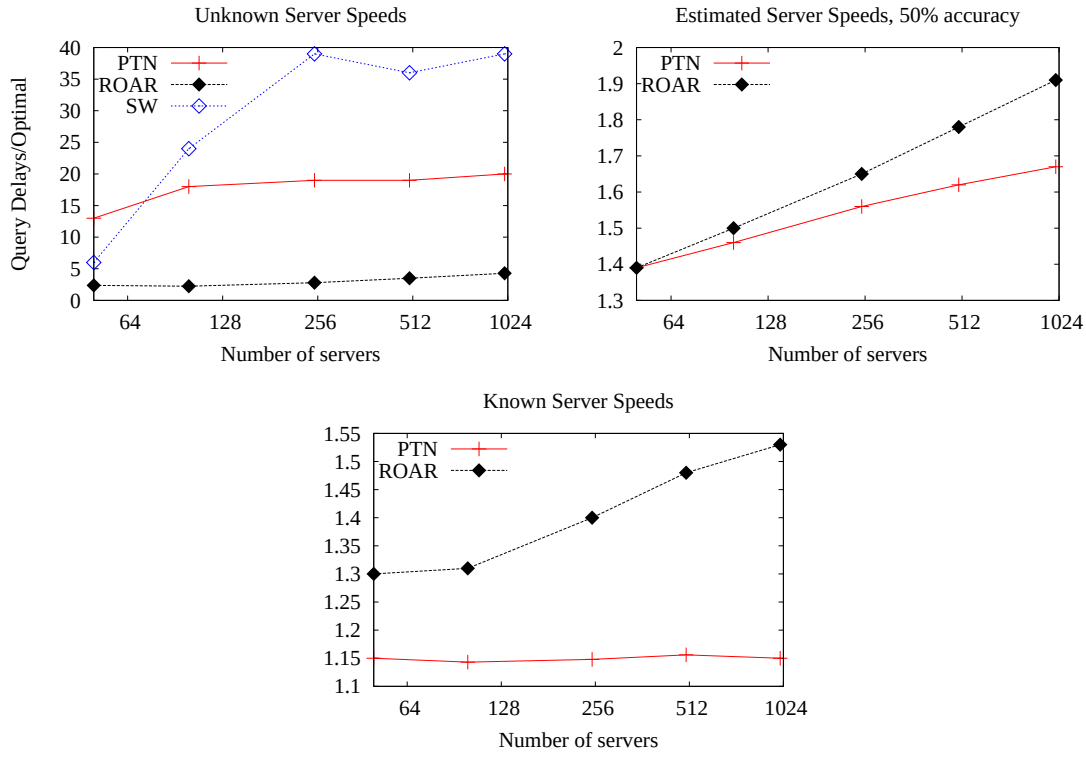


Figure 6.2: Variation of Query Delay with N

Vary n. We vary the number of servers from 50 to 1000. We report the average delay increase compared to the optimal as a function of n in Figure 6.2.

A first observation is that the shape of the delay curve as a function of p is similar across different network sizes; hence we can only report average delays, as opposed to exact query delays as a function of p. Further, this allows us to use smaller networks when running experiments to get similar results, while significantly reducing simulation time.

PTN with perfect knowledge does 15% worse than optimal, regardless of network size. ROAR does progressively worse as the network grows, but the slope is logarithmic and quite gentle.

With innacurate information, both ROAR and PTN struggle a bit more; the increase in query delay is still logarithmic with the network size, but is a bit gentler for PTN.

When the speeds are unknown, ROAR performs the best for any network size. PTN performs much worse in general, being 20 times slower than optimal. As before, SW is worst of all.

Varying Load. In our next experiment we vary the utilisation from 10% to 99%, running experiments for a network with 100 servers. We plot query delays relative to the optimal, as well as the number of times the algorithm was overloaded, out of the 50 runs (there is one run for each each value of p from 1 to $n/2$).

SW quickly becomes overloaded; at 30% load it cannot cope with the load for a quarter of the values of p. Again, we only show the SW curve on the first graph to help readability. SW has the same performance across all test scenarios.

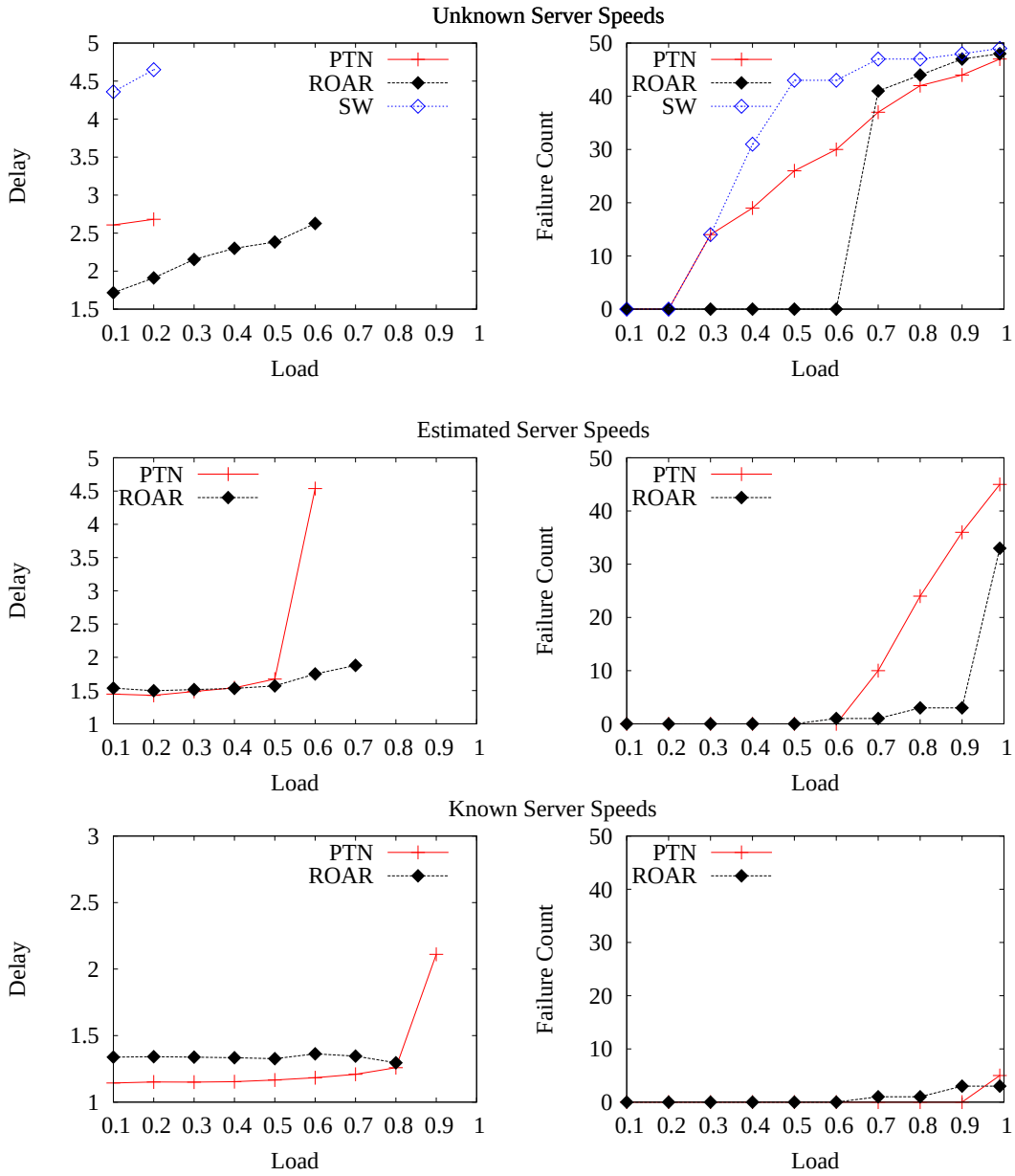


Figure 6.3: Variation of Query Delay with Load

In the case where server speeds are unknown, PTN is overloaded at 30% load. ROAR can cope with load up to 60%, being much more robust. As server speeds estimates increase in accuracy, PTN does a lot better, coping with loads of up to 60% for imperfect estimates, and 90% for perfect estimates. In contrast, ROAR has better performance for imperfect estimates, and similar performance when estimates are perfect.

Across all scenarios and nearly all load levels, ROAR either copes with the load or is overloaded for a small subset of the possible partitioning levels. When PTN is overloaded for a given configuration, it tends to be overloaded for many of the possible partitioning levels. Overall, ROAR can be used for a wider range of operating regimes.

Varying CPU capacity. We model server speeds by setting a lower and upper bound, and choosing values for individual nodes from a uniform distribution. The ratio between the lower and the upper bound is meant to capture the age difference of the servers, as newer servers are always faster, and servers have a finite lifetime of a few years (three years according to Greenberg et al. in [GHMP09]). Let us take an example: assuming Moore's Law holds and assuming a server lifetime of three years, the ratio between the speeds of the newest and the oldest servers is at most eight.

We vary the difference between the minimum and maximum speeds, starting from $mininimum = maximum$ and increase maximum until $64 \cdot mininimum$ (which corresponds to a six year period of adding servers). Query delays are presented in Fig. 6.4.

When the network is unoptimized, query delays increase linearly with the age difference between servers. This is to be expected, as weaker servers will take progressively more time to finish their tasks.

Using speed estimates to setup the network completely changes the shape of the curve. Surprisingly, the difference between PTN, ROAR and optimal quickly reaches a maximum. The default value we use for all other experiments ($max/min = 8$) is already on the flat part of the curve.

The reason for this flatness is simple: as enough servers become powerful enough, they will be able to service most of the queries. Both ROAR and PTN are able to use these servers; having an overall perfect allocation is less important, as the other servers will be mostly idle. As load increases, the flat part of the delay curve moves to the right.

Varying the setup phase's server speed estimation error. It is very important to understand how the algorithms behave with different quality server speed estimates. The extremes of perfect knowledge or no knowledge are unlikely to be of relevance in reality; somewhere in between these two will be the real operating point of the algorithms.

We vary the estimation error from 0 to 99%: if estimation error is e, the network setup phase will use an estimate of server speed x randomly chosen in the interval $[(1-e)x, (1+e)x]$. The delays of PTN and ROAR are plotted in Fig. 6.5.

The shape of the curve is not surprising, given the data points we have already observed in our other experiments: ROAR deals better with uncertainty than PTN since its query delays degrade more slowly as uncertainty grows.

To have a ground truth comparison between ROAR and PTN we need to know what the estimation errors will be in practice. However, these numbers will likely be different for different deployed systems, due to a different mix of background load competing with the algorithm, and may depend even on the exact type of search application being executed. In the end, all this graph tells us is that ROAR might be able to cope better with unpredictable server performance, which will make it preferable for deployment scenarios where background load cannot be controlled.

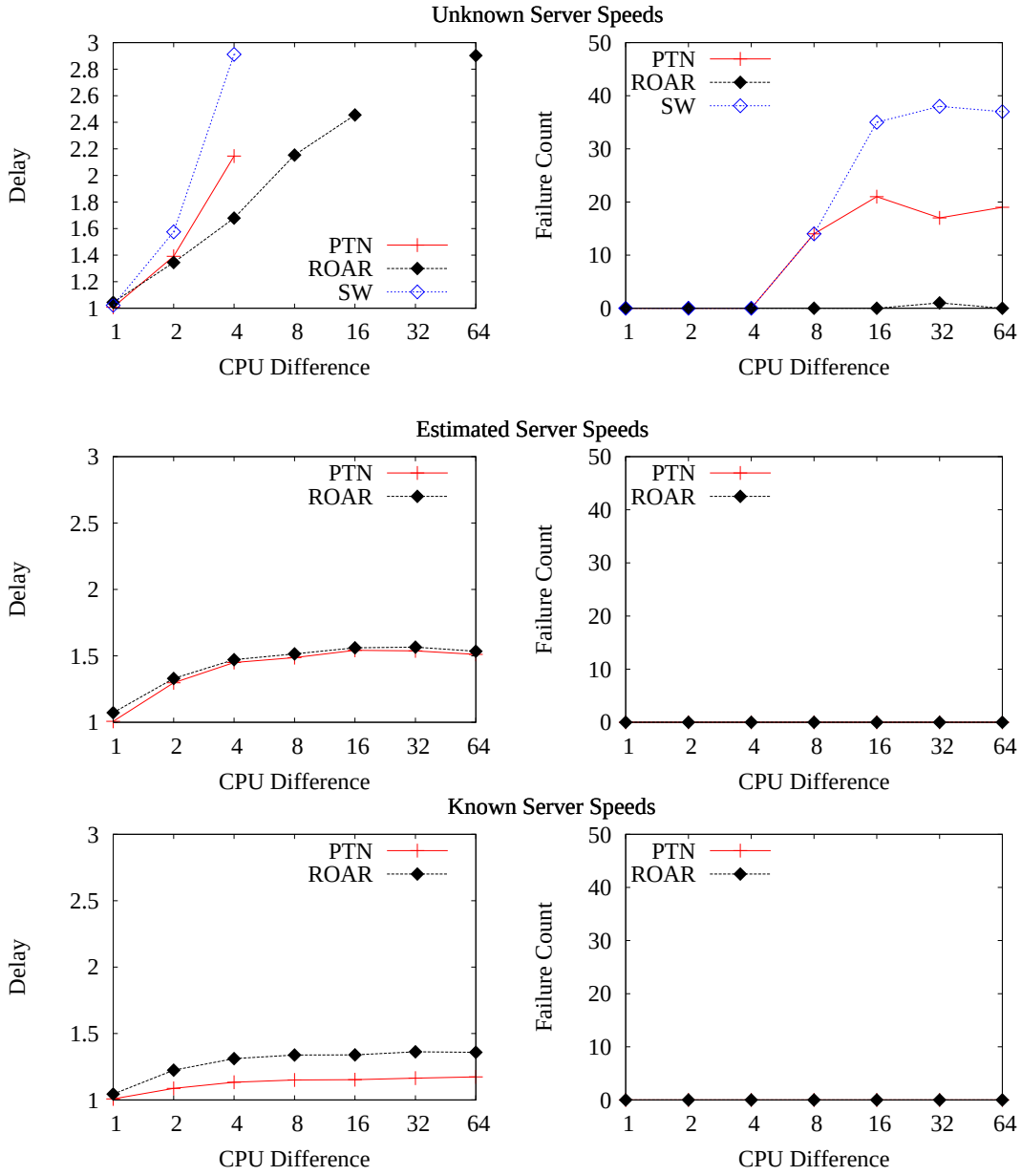


Figure 6.4: Variation of Query Delay with Server Heterogeneity

6.1.3 Query Delay Comparison when $p_Q > p$

Our previous experiments all assumed that each query is split into the smallest possible number of sub-queries, given the current replication level (i.e. $p_q = p$). For efficiency reasons, it may be possible to run the system at smaller values of p while providing query delay below the maximum threshold. However, absolute query delay directly depends on load, so short-lived load fluctuations around the mean might cause the algorithms to miss the delay targets occasionally. In such cases, a good approach is to temporarily increase p_q to reduce query delay.

To examine this effect, we setup a 100 server network with constant $p = 10$, and vary p_q from 10 to 50. We plot the absolute query delays in Fig. 6.6(a). The shapes of the delay curves for ROAR and SW are similar to the ones in the previous section, where we varied p.

A surprising finding is PTN's performance: the query delay curve has steps of length 10 (the value

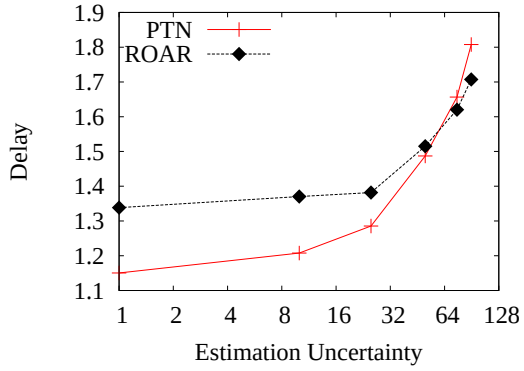
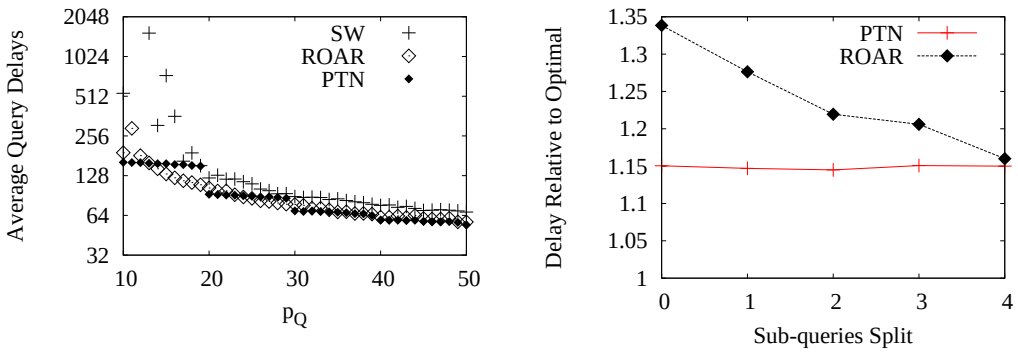


Figure 6.5: Algorithm Performance with Different Server Speed Estimation Errors



(a) Varying p_Q from 10 to 50 (p=10)

(b) Small increase in p_Q for ROAR only

Figure 6.6: Increasing p_Q and its effects on the algorithms

of p). Because of these, PTN's delay is worse than ROAR's for most of the time. The explanation is straightforward: PTN sets up 10 clusters, and it only benefits from increasing p_q when it can split the sub-query destined for each of these clusters. Hence, the performance is best when p divides p_q, and gets progressively worse when it does not. In contrast, ROAR and SW are much more flexible in query partitioning, and can effectively split using any values of p_q.

As ROAR is so good at dealing with arbitrary p_q, a natural question arises: can we effectively use this ability to reduce query delays? Rather than splitting a query into $p_q > p$ sub-queries, we take a different approach: we split into p sub-queries, and then split again the sub-query that would finish last. We repeat this step a small number of times.

The delays of ROAR and PTN (used as a baseline) are presented in Fig. 6.6(b). By increasing p_q by 2, query delays for ROAR decrease to 20% of optimal, being very close to PTN's performance. Further increasing p_q bring more benefits, but also increases costs due to more sub-queries, and may be undesirable. This result is very useful in practice: selectively splitting the sub-query that would finish last basically aligns ROAR's query delays to those of PTN.

Note that all the experiments in this section were run assuming perfect knowledge of server speeds; the results are qualitatively similar for imperfect knowledge. We omit these results for conciseness of

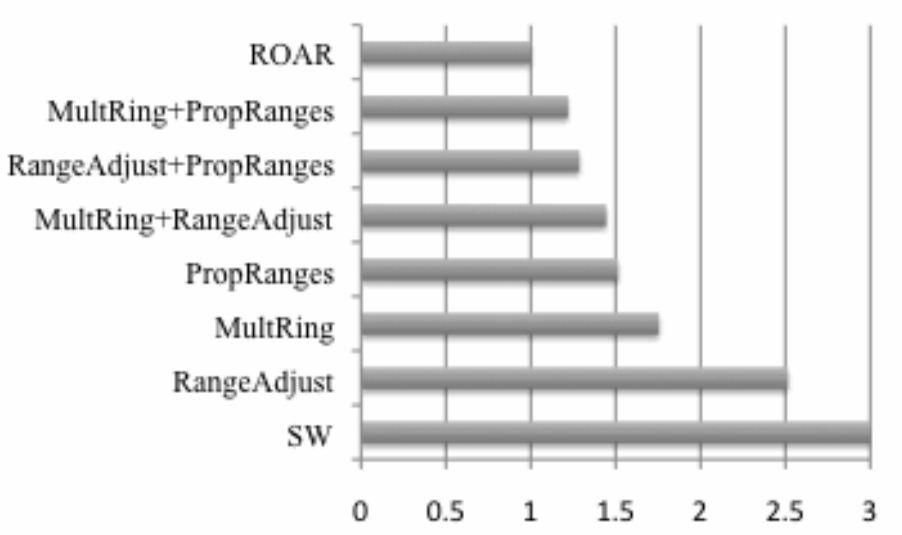


Figure 6.7: Effects of ROAR Mechanisms on Performance

presentation.

6.1.4 Analysis of ROAR Mechanisms

ROAR includes a few mechanisms that differentiate it from SW:

- **Assigning server ranges proportional to processing speed** is mainly intended for better load balancing, but also reduces query delays as more powerful servers can run more queries. This mechanism uses estimates of server speeds, and its effectiveness is directly related to the estimation accuracy.

- **Using multiple rings** gives ROAR the power of two choices for scheduling queries. Each query can choose a configuration of servers out of $2^{p-1}r$ possible, whereas SW only has r possible configurations to choose from.

- **Adjustment of sub-query ranges** runs after the scheduler has assigned each sub-query, attempting to move work away from the most loaded node.

Together, these mechanisms make ROAR delay performance comparable to PTN, and orders of magnitude better than SW. It is equally interesting to tease apart these end-to-end numbers, to understand how each mechanism contributes to overall performance.

We use our default setup to run experiments using all possible combinations of mechanisms. We run basic SW, each mechanism enabled on its own, then all combinations of two mechanisms, and finally all three mechanisms.

The results are presented in Figure 6.7, with query delays relative to ROAR. The delay for SW cannot be rigorously calculated, as SW is overloaded for some values of p. The bar is there to provide a baseline, albeit a hypothetical one.

Each of the mechanisms individually solves the overload problem, and reduces average query delay. The biggest impact is due to Proportional Ranges. Multiple Rings provide similar benefits, while Range

Adjustment is not as effective. The results for Range Adjustment are surprisingly good, considering it is just a local O(1) heuristic that softens the peak of sub-query delays.

The performance of Proportional Ranges is also surprisingly good; however it will not be as good in practice. The experiments we ran assume perfect information on server speeds; our earlier experiments have shown how imperfect information affects the performance of ROAR, and indirectly, of the Proportional Ranges mechanism.

Taken pairwise, the mechanisms incur progressively lower delays. The best combination, as expected, is Proportional Ranges and Multiple Rings. Finally, all three mechanisms work together harmoniously in ROAR, giving the overall best performance.

These results are not necessarily surprising: Range Adjustment runs last and never makes things worse. It strictly increases performance in any configuration, so it should function well with any other optimisations. Multiple Rings literally takes a single ROAR ring and splits it in two, bringing very little overhead yet providing two possibilities for placing each sub-query. Proportional ranges complements multiple rings, as it applies to each individual ring. The synergy and increased benefits come from their combination. If any of the mechanisms mis-functions (as Proportional Ranges does with incomplete information), the other mechanisms keep performance at good levels.

6.2 Fault Tolerance

Data center algorithms rely on stable populations of servers to perform their tasks. This is in stark contrast with larger scale peer-to-peer search algorithms (like BubbleStorm [TKLB07]) where high server churn is the norm, and needs to be taken into account in algorithm design. In ROAR stable server populations allow optimisations like proportional ranges to work, and the single administration allows using a centralised membership server.

Nevertheless, servers fail even in data centers. The higher the number of servers used, the higher the overall probability that at least one or a few servers have failed at any point in time. We want to compare PTN and ROAR fault tolerance, and a useful starting point is the study of Yu et al. on availability of multi-object operations [YGN06].

Yu et al. study three classes of algorithms - SW, RAND and PTN - and two classes of applications requiring either all objects (strict operations) or a fraction of objects (loose operations) to be read. They find that for strict operations PTN gives best availability, with SW second and RAND third. For loose operations, the order is reversed, with RAND best, SW second and PTN third. The main insight is that PTN attempts to maximise inter-object correlation, by storing the same groups of objects onto different machines, and that is why it gives best availability when all objects are required. RAND does the opposite, randomly placing replicas on machines, thus minimising inter-object correlation; RAND does better when operations are loose.

RAND is not feasible to use for distributed rendezvous operations because of increased costs in storage and/or queries. We focus the analysis on SW, PTN and ROAR, and restrict our analysis to the more demanding strict operations.

We fix $r = 3$ ($p = 34$) and use our default setup with 100 servers and 8x CPU difference between

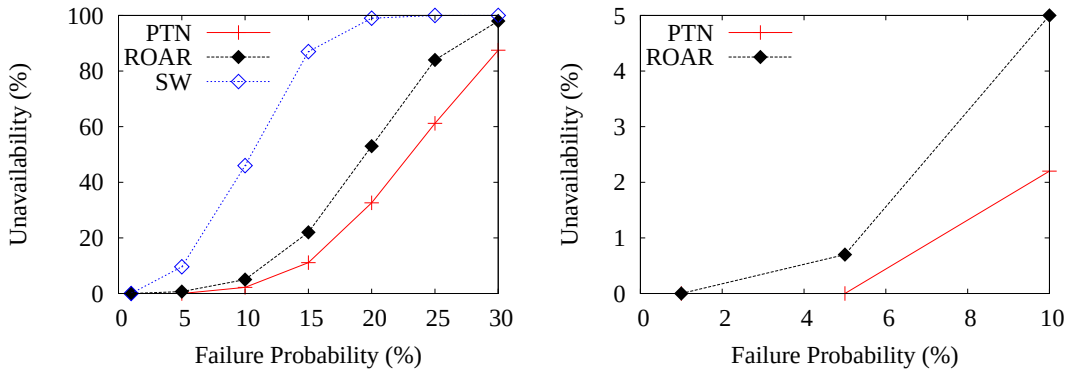


Figure 6.8: Algorithm Unavailability Comparison for Strict Operations

servers. Each experiment selects a server failure probability and runs many iterations, in each iteration randomly failing servers across the network. It then schedules one query with $p_q = 100$ and checks to see if all sub-queries were successfully scheduled; if so, unavailability is 0%. Otherwise, if there are sub-queries that can not be executed, unavailability is 100% for that configuration. We report the average unavailability across the 1000 experiments.

ROAR stores a few more replicas than PTN and SW, because of the "rounding up" effect of object range intersection with server ranges. The net increase is on average 0.5 replicas per object, and it does not really matter unless r is small. For these experiments, however, this difference does matter. To make the unavailability analysis fair we run ROAR with $p = 40$ instead of $p = 34$. This makes the average replication rate the same for all algorithms.

Unavailability as a function of failure rate is presented in Figure 6.8. In the left-hand side plot failure rates grow up to 30%, and consequently unavailability increases significantly for all the algorithms. The results confirm Yu et al.'s study, showing that PTN gives much better availability for strict operations. From a fault tolerance perspective, ROAR's multiple rings make it behave like a hybrid between SW and PTN. ROAR has worse availability than PTN, and much better availability than SW.

The data points for more than 10% loss rate are useful to give insights into the fundamental algorithm behaviour, but are not realistic failure rates for data centers. In particular, annual disk failure rates are around 3% ([SG07]), so the weekly average number of failures is very small (a week seems to be the upper bound to getting servers/disks replaced in current data centers). From this perspective, all the algorithms have perfect availability, so the comparison is irrelevant in practice.

In the right hand side plot of Figure 6.8 we zoom in on the results in the 0%-10% failure rate, and only compare PTN and ROAR. ROAR has 1% unavailability for 5% failures and 5% for 10% failures. In comparison PTN unavailability is 0% and 2% respectively. However, the random server failures model assumes there are no correlated failures.

Another noteworthy point to consider are switch failures. Data centers typically have three tiers of switches, and switches at aggregation and core levels are typically redundant, as multiple paths exist between any source-destination pair in the network [AFLV08, GLL+09, Gre09]. Hence, failures of these switches will typically mean no servers are disconnected; however less overall throughput is available.

	PTN	SW	ROAR
Store Object	r	r	$r \geq 2$
Execute Query	p	p	p
Increase p	$\frac{N \cdot D}{(p+1)^2}$	0	0
Decrease p	$\frac{2 \cdot N \cdot D}{p(p-1)}$	$\frac{N \cdot D}{p(p-1)}$	$\frac{N \cdot D}{p(p-1)}$
Increase r	$\sim 2D$	D	D
Decrease r	$\simeq D$	0	0

Table 6.2: Bandwidth consumption comparison (messages per operation)

Top-of-rack (ToR) switch failures are a different story. In typical scenarios there is no redundancy, so a single switch failure can take-out 20 to 40 servers. In a 1000-server network, these correspond to 2%-4% of servers failing. ToR switch failures are a valid concern and will cause large-scale failures and disruption for all algorithms. Possible solutions include replicating the data more, or doubling the number of ToR switches and multihoming each server to two ToR switches.

6.3 Changing the p/r tradeoff

To get a full comparison of the algorithms, we must understand how good they are at changing p at runtime. We first focus on the issue of how much bandwidth each of the algorithms consumes when performing their various operations; Table 6.2 lists the number of messages sent per operation. The costs shown in the first two rows of the table, concerning storing objects and executing queries, are obvious from the algorithms. It is worth noting that the cost for executing queries is a lower bound corresponding to the case when queries are sent to exactly p servers.

Let D be the total aggregate size of the unique objects. When changing p PTN incurs the highest overhead. The calculations are simple and fall through from the way PTN operates. PTN increases p by removing $\frac{D}{p+1}$ objects from existing clusters and replicating them on a new cluster containing roughly $n/(p+1)$ servers (which preserves load balancing). When p is decremented the same reasoning applies: first objects from the cluster to be destroyed are stored onto existing clusters, and then the servers join the remaining clusters, storing all the objects in those clusters.

We also list the amortised bandwidth cost to increment/decrement r, computed as the cost to increment/decrement p divided by the change in r this brings. Note that in reality it is not always possible to increment or decrement r: step changes in p may change r by much more than 1. However, this exercise helps us understand better how close to the optimal these algorithms come. Both SW and ROAR copy less data when changing p (and thus r), and are optimal from this point of view. PTN is suboptimal: it copies D more data when both incrementing and decrementing r.

Convergence Time. We now turn to convergence time, which we define as the time from when the algorithm decides to change p to when the algorithm finishes copying the necessary replicas in order to ensure routing correctness.

SW and ROAR equally spread the copying of new objects across all servers (assuming roughly equal ranges). PTN, however, places more load on some servers. When p is decreased (and thus a cluster is destroyed), the servers from this cluster join a new cluster. Here, each server needs to copy *all* the data

in the new cluster, roughly $D/p - 1$. In comparison, in SW each server only needs to copy $D/p(p-1)$. This means that the time required to copy the data needed to decrease p for PTN is p times larger than in SW.

To understand if this matters, we analyse two simple scenarios. Assume Google wishes to decrease p; reports place $p \simeq 1000$ [Dea], and each machine stores 2GB of data (so the whole dataset is $D =$ 2TB). It follows that each node will have to copy roughly 2GB when it joins the new cluster. This takes roughly 20 seconds on gigabit links. In contrast, the time needed by SW and ROAR is 1000 times smaller, namely 20 milliseconds.

A high-volume storage application would store on each server enough data to fill a sizeable portion of its hard drive, say 100GB at least. Let us assume the dataset is still $2TB$ in size, and thus $p = 20$. To decrease p, PTN needs to copy roughly 100GB, which would take 20 minutes on a gigabit link; the same would take a single minute for SW or ROAR.

Distributed State. The fact that servers have equal roles in changing p in SW and ROAR also reduces the complexity needed to implement the change. Essentially, during the change all servers are in either state p or $p-1$, and can be used as soon as they switch. The scheduling algorithm seamlessly operates in this transient state, and the costs are almost negligible. There is no need to distributedly agree on what the state of the system is. This has implications for the membership and front-end servers for PTN, which must be tightly synchronised. First, the servers must agree on which cluster should be destroyed, which requires running a distributed coordination algorithm like Paxos [Lam01].

Secondly, let us consider how servers actually switch between clusters. They will first copy all the data needed in their new cluster, switch to the new cluster by informing the scheduling servers, and drop the data in the old cluster. *When* does this switch happen? One strategy is to switch as soon as possible, i.e. when the data has been copied; this would create a bottleneck on the cluster to be destroyed, as the same number of queries would be handled by fewer servers. In the worst case, a single server is left to handle all the load of the cluster. To avoid this situation, some queries for the old cluster could be redirected to remaining clusters, as these also store that data. This temporarily increases the load of the old clusters, which now serve the whole dataset D instead of $D(p-1)/p$.

Thus, to reduce capacity loss during the change, the best option seems to be to switch all the servers from the old cluster to their new clusters at once. This again requires distributed coordination and appears difficult to get right at large scales.

6.4 Comparison Conclusions

PTN has better availability at high failure rates and better query delays when server speed can be perfectly estimated. In data centers, however, failure rates are really low. Additionally, server speed is very difficult to estimate perfectly—as we will also show in our experiments.

In realistic scenarios PTN and ROAR offer comparable delay and high availability. The key benefit of ROAR is that it allows more flexibility. ROAR systems can use small increases to p_q to gain immediate benefits in query delay; PTN is more sluggish to respond to such increases, as p_q needs to be a multiple

of p to get benefits. ROAR changes the ratio between p and r seamlessly and with optimal bandwidth consumption, while PTN unequally loads servers during the change and transfers significantly more bytes. SW's performance is far inferior in all respects.

Can we use this flexibility for end-to-end performance increases, and cost decreases? We answer this question positively in our experimental evaluation in Chapter 7.

Chapter 7

Experimental Evaluation

To evaluate ROAR we built PPS on top of ROAR and deployed the system on 50 servers on the Hen testbed at UCL and on the Amazon Elastic Compute Cloud. This evaluation has three major goals. First, we wish to see how p impacts the properties of the system, including the average query delay, throughput, and utilization. This gives insight into the types of values that are appropriate for p in practice, and will 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? Is the frontend a scaling bottleneck? How well do the load balancing mechanisms work?

Third, we complete the evaluation with a head-to-head delay comparison of ROAR and PTN. We want to know how the two algorithms compare in realistic conditions, with inherent delay variability due to OS runtimes and changing network conditions. In the process we will also cross-validate the simulation results and gain insights into the predictability of runtimes in real systems.

7.1 Experimental Setup

We implemented ROAR in Java. The implementation has approximately 8 thousand lines of code (LOC), including code for the ROAR servers (5,000 LOC), the membership server (600 LOC) and the frontend query manager (1,500 LOC).

We mainly used the Hen testbed at UCL to test ROAR. The testbed contains approximately 100 net-booted servers from various vendors and purchased at different times. This heterogeneity helps our evaluation, providing a realistic distribution of server performance. Table 7.1 provides a summary of the models used in the experiments.

The Hen testbed is used by many researchers concurrently, with each machine being exclusively used by one user at a time, hence we were not able to use all the machines simulatenously. Our experiments were run on approxiamtely half of the machines which we acquired for relatively short periods of time (one to a few days). The set of machines we used changed constantly with each different experiment, providing confidence that the obtained results are not an artefact of the experimental setup. To evaluate ROAR at scale we briefly rented out 1000 servers from Amazon's Elastic Compute Cloud and ran experiments. Our findings are presented in Section 7.7.

Vendor	Model	Processor(s)	Memory	Disk
Sun	X4100	AMD Opteron 248 @2GHz	2GB	SEAGATE ST973401LSUN72G
Dell	PowerEdge 1850	Intel Xeon @3.00GHz	2GB	SEAGATE ST373207LC
Dell	PowerEdge 1950	2 dual-core Intel Xeon 5150 @2.66GHz	2GB	MAXTOR ATLAS10K5
Dell	PowerEdge 2950	2 quad-core Intel Xeon X5355 @2.66GHz	8GB	DELL PERC 5/i

Table 7.1: Server Models Used in Experimental Evaluation

7.2 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 Privacy Preserving Search and was extensively discussed in chapter 5. 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 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, as do the two versions of PPS we used: low memory (PPS_LM) and low cpu (PPS_LC). 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 does not 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 disk bandwidth. The different versions of PPS have quite different fixed costs, as we would also expect when comparing regular web search with webmail search.

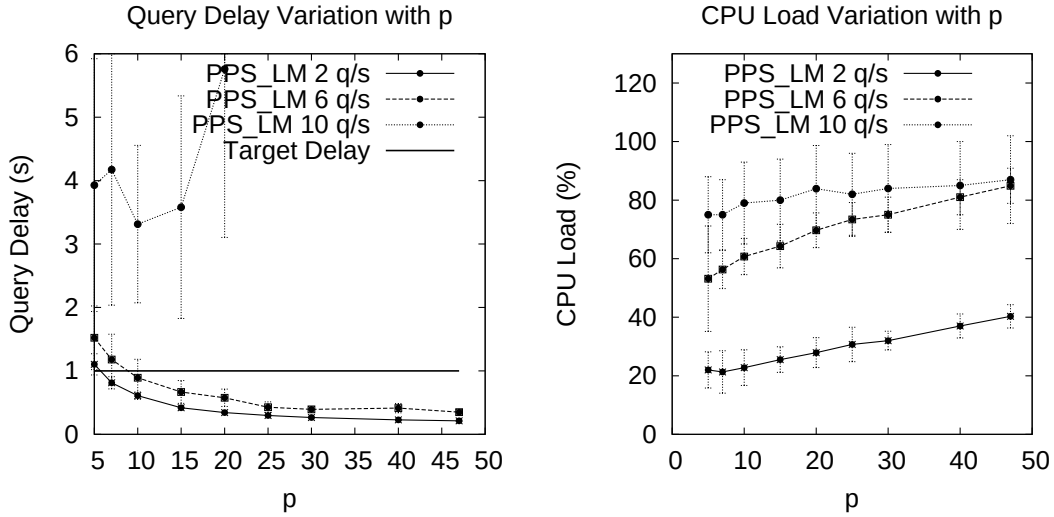


Figure 7.1: Effect of p on system performance with PPS_LM

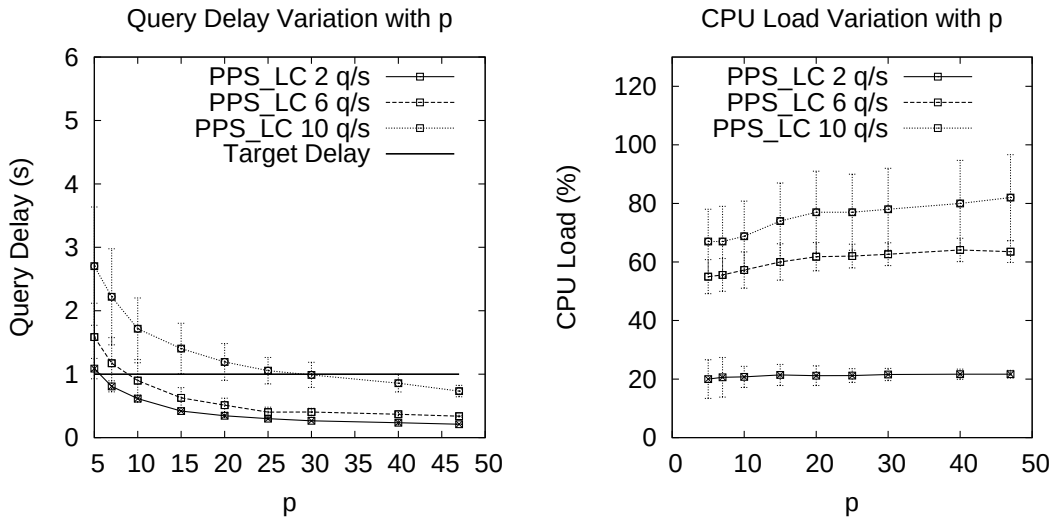


Figure 7.2: Effect of p on system performance with PPS_LC

7.3 Basic Tradeoff

To examine how p impacts query delay and throughput we used 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 50 testbed servers, 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.

Our initial setup used mostly the slow servers (X4100 and Dell 1850), and ran from the buffer cache. For PPS, it is unlikely that user data will be in memory when a query arrives, so loading from disk is very likely. However, hard disk drives are being replaced with solid state drives in enterprise deployments, as these offer significantly more performance than state of the art hard drives. Any PPS deployment will likely use solid state drives, thus performance numbers from disk-bound systems will not be representative. To "simulate" faster disks we relied on the OS buffer cache.

To allow a single server to search its part of the index in one second, we started with a value of $p = 5$,

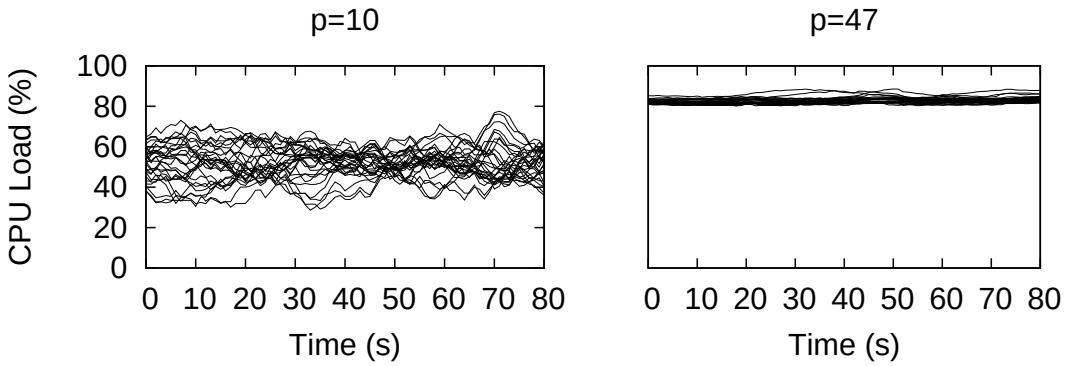


Figure 7.3: Average CPU load for each node

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.

7.3.1 Query Latencies Decrease with p

The results are shown in Figures 7.1 and 7.2 for PPS_LM and PPS_LC respectively. 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.

7.3.2 Query Overheads Increase with p

The same figures (right hand plots) show 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.

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 7.3 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

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

Table 7.2: Energy Savings running at $p = 5$ instead of $p = 47$

much more as queries come and go. When $p = 47$ there are few idle times and load is heavily and constant.

7.3.3 Higher Overheads=Wasted Resources

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 7.2). Comparing $p = 5$ with $p = 47$, our newer servers[1] were measured to consume 18W more with PPS_LC and 50W more with PPS_LM. Our older servers[2] 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.

Scaling up these numbers, if we had a testbed of 47 latest-generation servers, the energy gain for running with a small value of p would range from 0.9KW to 3KW; at current electricity prices this would represent increased operating costs of $600 to $2000 per year. In a moderately sized data-center with 30000 last-generation servers, the cost increase due to the value of p would be between 0.4 and 1.2 million dollars.

Each query requires a disk seek then a read of 250MB of contiguous data. When disk-bound, increasing p not only increases CPU overheads but also increases the ratio of seeks to reads, 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. In a disk-bound system using a higher p could reduce maximum throughput by 10%.

When p increases, the workload on each server becomes more fragmented; smaller values of p create longer tasks and, on average, longer idle periods. These idle periods, if long enough ($> 1s$) could be exploited to save energy by spinning down hard drives. Although this is not feasible for server-class drives (spin-up/down time is on the order of tens of seconds), it is feasible for laptop-class drives (with spin-up/down times of well under a second). Concretely, for the workload of 2 queries per second, it should be possible to spin down the drives for 60% of the time, saving between 6 and 10W per server. This technique is similar in concept to “write offloading”, a technique that increases inter request gaps allowing disks to be spun down for longer periods [NDR08].

Finally, the bandwidth required to run a single query increases proportionally[3] with p. This does not create a sizeable impact on energy consumption, but will increase usage of the scarce cross-section bandwidth.

In summary, increasing p above the minimum needed to satisfy the required delay bounds increases

[1] Dell PowerEdge 1950 and 2950

[2] Sun X4100 and Dell 1850

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

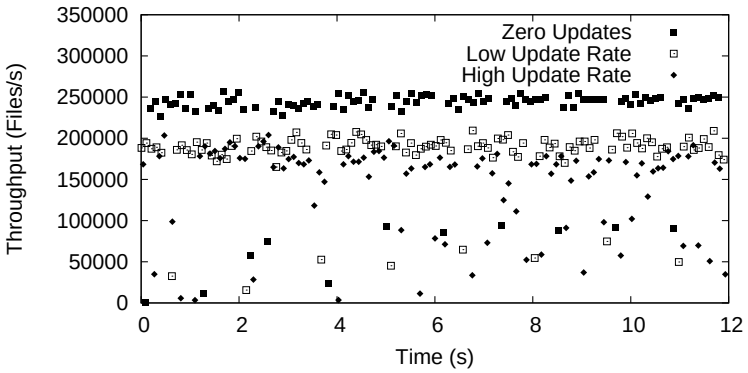


Figure 7.4: Effect of updates on server throughput

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.

7.3.4 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 7.4 shows 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.

It is worth noting that this effect is not unique to ROAR: with any distributed rendezvous scheme the operator needs to consider using a larger p than might otherwise be required if the data replication costs start to become non-negligible.

7.3.5 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 have 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. When the load changes, we will need to reconfigure p and r to match it.

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.

7.4 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

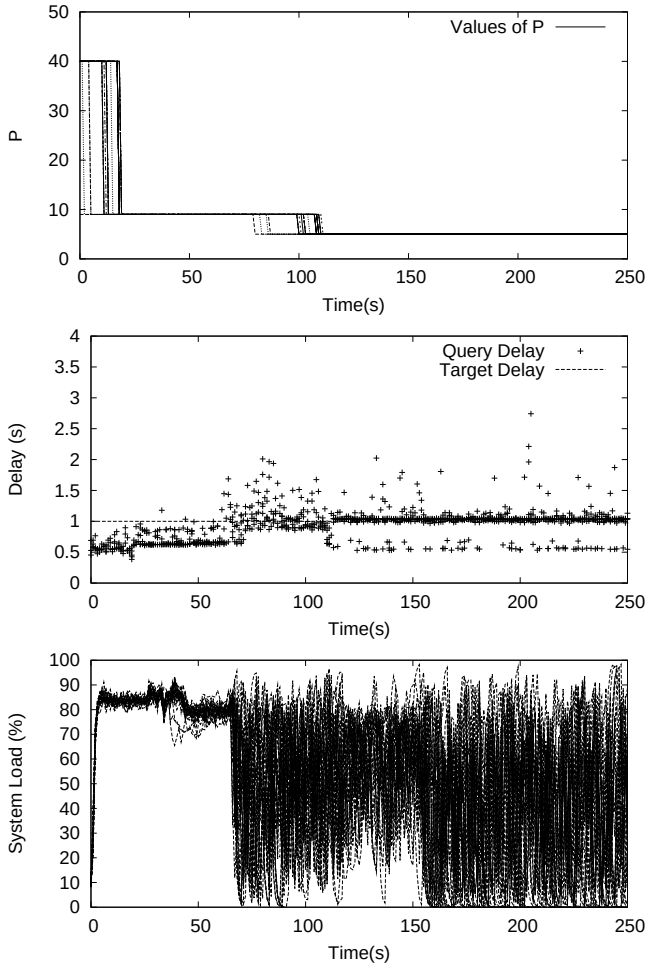


Figure 7.5: ROAR Changing p Dynamically

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 7.5.

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

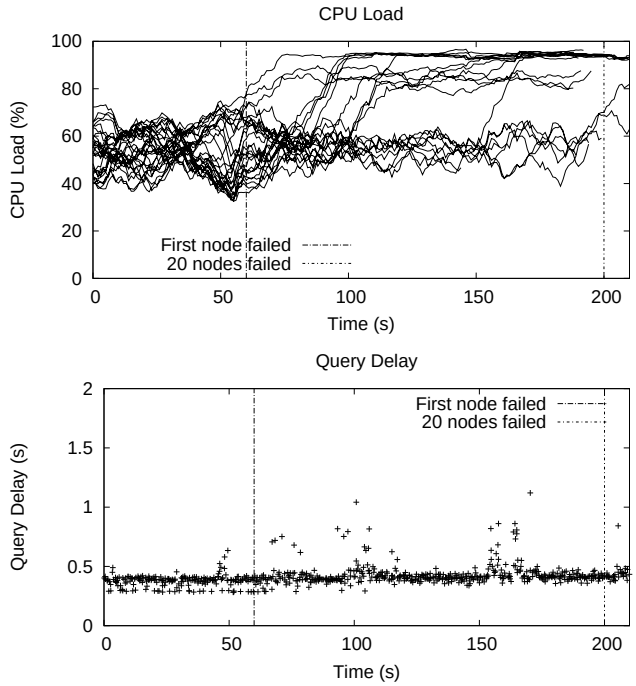


Figure 7.6: Effects of 20 Node Failures on ROAR

to be increased to reduce the memory strain on each server (this seems to be a constraint in 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 in p so long as an optimization function can be defined that captures the relevant constraints.

7.5 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 7.6. 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

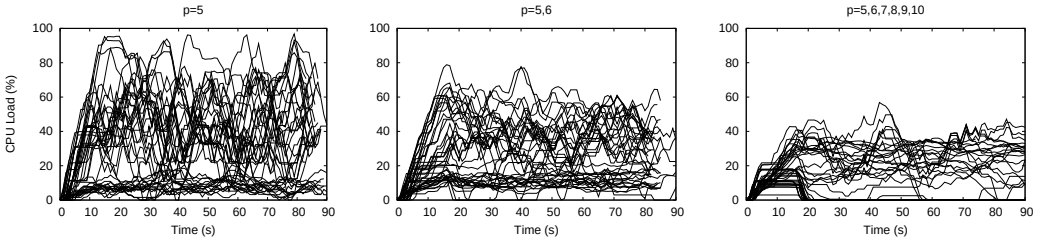


Figure 7.7: Fast Load Balancing with $p_q > p$

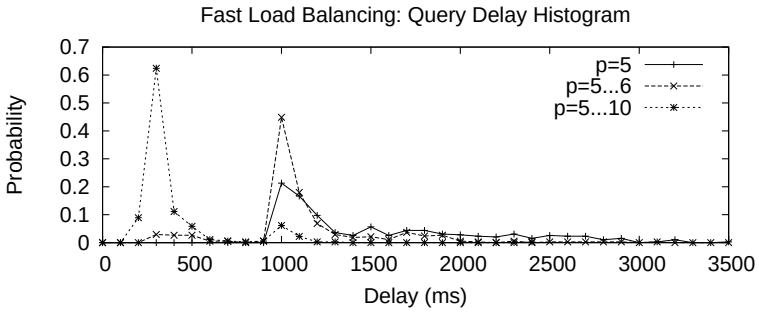


Figure 7.8: Delay Distribution with Fast Load Balancing when using $p_q > p$

7.4.

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.

7.6 Load Balancing

The previous experiments were conducted with mostly 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 (Section 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 7.8 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 10 if needed. It is clear that this mechanism is effective at moving load onto the faster servers.

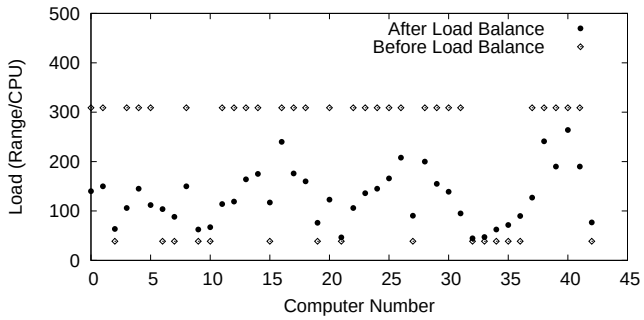


Figure 7.9: Range Load Balancing

Figure 7.7 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. 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. 7.9). 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. 7.10. 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.

7.7 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 [Ama]. 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 of our slower HEN servers because the CPU is slower: a query of one million metadata items takes eight seconds.

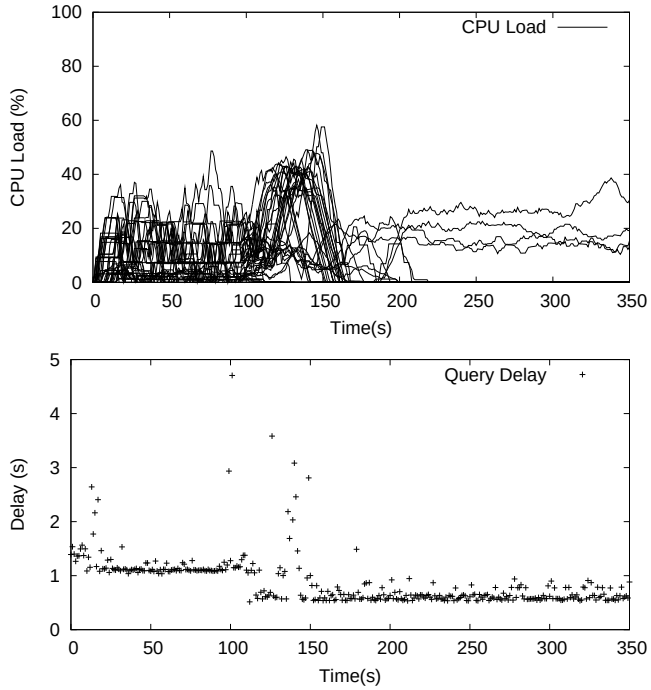


Figure 7.10: Effects of Range Load Balancing

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 7.3 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 $n \log p$ 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.

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[4] increases from 1.2 to 4 when p goes from 100 to 1000.

To nail the cause of high delays observed, Figure 7.11 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.

The answer is the synchronization of the query replies, coupled with small buffers at output-buffered switches, a problem that is known as TCP incast [VPS+09]. When p is large many servers will reply

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

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 7.3: ROAR performance running on 1000 servers in EC2

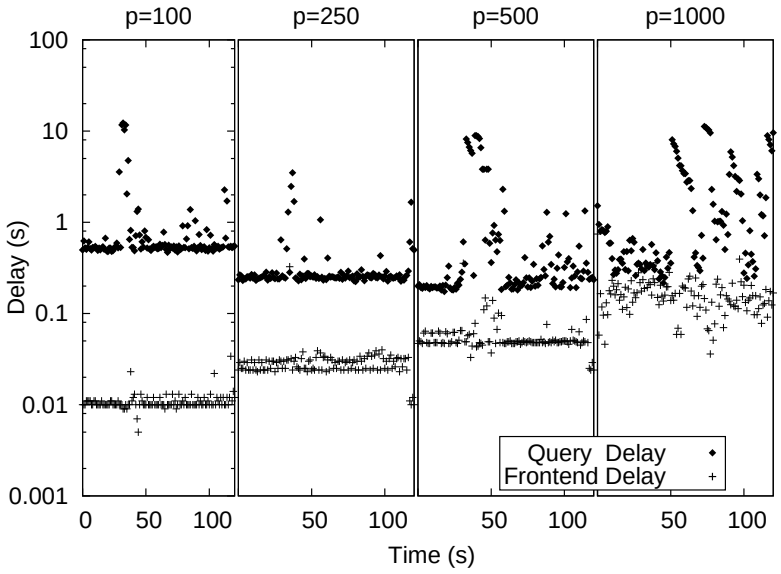


Figure 7.11: Delay Breakdown as seen at Frontend Server

at roughly the same time, and all these replies will arrive simultaneously at the switch, overflowing the output port of the link going to the frontend server. 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 7.11 show losses are bursty and tend to synchronize over many timeouts, escalating the problem,

A simple fix for TCP incast is to eliminate the RTO lower bound and compute nanosecond accurate TCP timers [VPS+09]. Its unclear that this will solve the synchronization of retransmissions. A simple application-level alternative might mitigate these losses: the frontend should resend unfinished query parts as soon as most of the query has completed. At least for our application, this implies that UDP might be a more appropriate transport for ROAR. Even without any of these fixes, controlling p gives a simple way to mitigate the effects of incast, and to re-adapt the system when the network configuration changes.

We were not able to notice incast issues in our Hen deployment because the Force10 switch Hen uses is massively overbuffered, having 2.4MB of buffering per port. Looking at the future, the industry

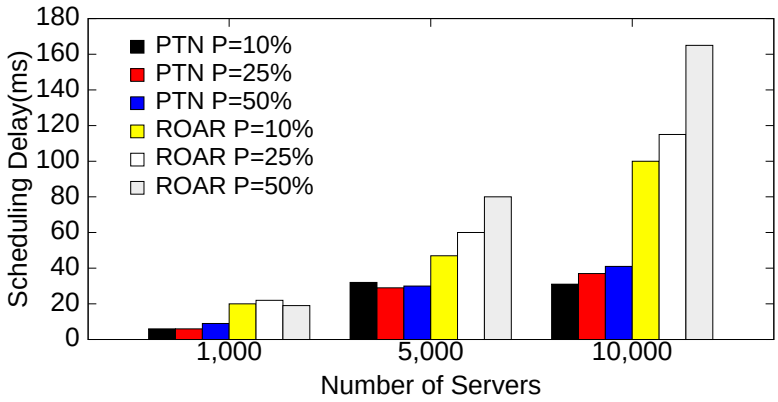


Figure 7.12: Frontend Scheduling Delay for PTN and ROAR

has already moved to solve the TCP incast issue by sharing the buffers across ports. This does solve incast, but creates interference between cross traffic. Thus, the use of RED [FJ93] has been recently proposed to manage queue sizes [AGM+10].

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.

7.8 Frontend Scheduling Performance

Our large scale deployment gives one data point showing that the frontend can scale up to a thousand nodes. Here we go further and examine its performance at larger scale, and in comparison with the simpler PTN scheduling routine.

To perform the comparison, the membership server was changed to create "fake" servers with random ranges. A few thousand queries were then scheduled by the frontend, and the scheduling delay was recorded. In contrast to the large scale experiment, the frontend numbers here do not account for the delay to serialize and send all the sub-queries, focusing only on the time needed to schedule the query.

Results are presented in Fig. 7.12 for one, five and ten thousand servers, and three values of p for each. In Section 4.8 we found the algorithm complexity to be $O(n)$ for PTN and $O(n \log p)$ for ROAR. The experimental results show that in practice the $\log P$ factor results in two to four times increase in scheduling delay for ROAR compared to PTN.

Our prototype is written in Java and unoptimized, yet the absolute numbers are encouraging: scheduling on 5000 servers (the estimated size of a Google search cluster) takes on average 62ms compared to PTN's 30ms. Batch-scheduling simultaneous queries is a very simple technique that can be used to reduce query delays, and is quite effective when it is needed most: when queries are frequent. When using batches of 5 queries, scheduling delay would drop to around 10ms.

In its current incarnation, our prototype running on a single core can handle about 16 queries per second without batch scheduling, and 100 queries per second with batching. The scheduler is, however,

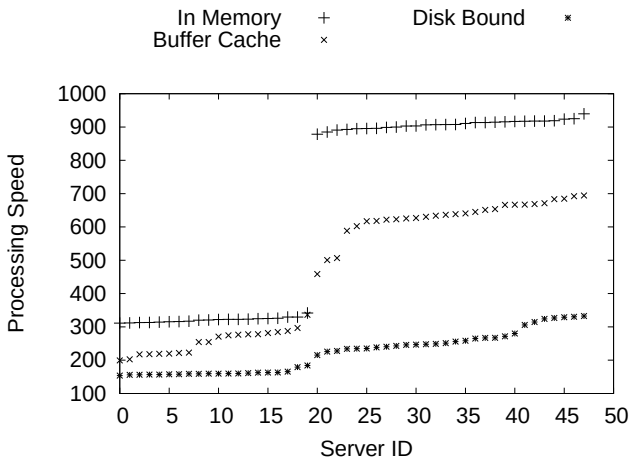


Figure 7.13: Observed Server Processing Speeds

easily paralelizable to many cores or indeed to many machines, and can easily be scaled to support very high query rates.

7.9 Query Delay Comparison: ROAR vs. PTN

We have so far shown that ROAR can easily adjust p while queries are running and this brings end-to-end benefits such as reduced power consumption and increased throughput. One lingering question is, however, how do ROAR's query delays compare to PTN's? Our analytical evaluation offered insights into when ROAR is worse, better or equal to PTN, but we still need to see how the algorithms perform in practice.

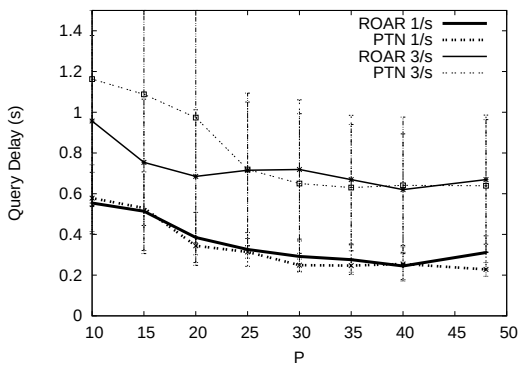
To gain a better understanding of the differences, we implemented PTN too and performed a head-to-head comparison of the two algorithms. Implementing PTN was relatively straightforward, given the code base we had for ROAR: the biggest changes were made to the membership server (that now creates clustered ranges for nodes) and to the frontend server.

All our previous experiments ran from the buffer cache, as that was the setup that was closest to real world deployment. Here, however, we wish to compare the algorithms in a range of operating regimes, and see the differences. Besides the buffer cache, we ran experiments where data was read from disk (disk-bound), and one where data was already in the memory-cache. These three represent the spectrum of operating regimes for PPS; reality will be somewhere in between.

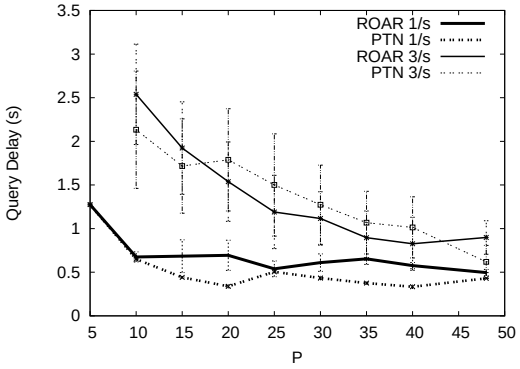
Running experiments with data from hard disks is tricky: unless huge amounts of data are read, the OS buffer cache kicks in and reads do not reach the disk anymore [5]. To avoid the buffer cache we create 200 users each with 1M objects. In total the metadata has 40GB, which are partitioned and replicated on the servers such that $p = 10$.

For the buffer cache and in memory experiments we create a more stressing test environment, with 5 million metadata being searched by each query. On our fastest machines, running this query would

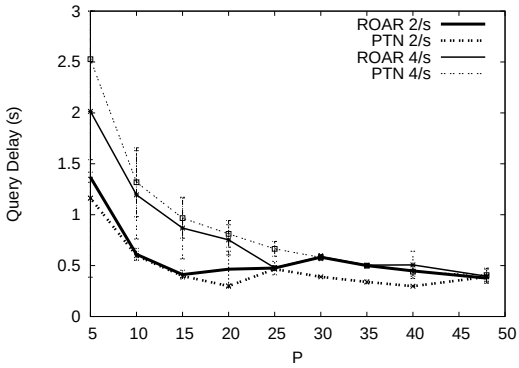
[5] Manually clearing the buffer cache is possible but tricky because our servers netmount their root filesystem; periodically clearing the buffers for all 50 servers created such a high load that it brought down our NFS server.



(a) Disk-Bound, 1M objects



(b) Buffer-Cache, 5M objects



(c) In Memory, 5M objects

Figure 7.14: Query Delay Comparison ROAR vs. PTN

take well over 4s.

We show the sorted server processing speeds for the three scenarios in Figure 7.13. Disk-bound is, as expected, slowest of all and has three different plateaus, corresponding to the three hard drive models in our slow, intermediate and fast servers. The speed difference between the fastest and slowest server is 2x. In memory processing is significantly faster, and has a bimodal distribution, with slower servers being 3 times slower than faster ones. Finally, loading from the buffer cache is in between, with a 3.5x difference between the fastest and slowest machines. These values validate our choices in the analytical evaluation, where we used a 4x difference between the slowest and fastest servers.

Query delay results are presented in Figure 7.14. Each graph shows the algorithms running under light load (1 or 2 queries/s) and moderate load (3-4 queries/s). ROAR and PTN have similar behavior in all scenarios, with PTN outperforming ROAR when load is low (1 or 2 queries/s) by 5% to 40%. on average. Under moderate load, ROAR outperforms PTN by 5% to 15%. The explanation is simple: when load is low machine performance is highly predictable, and PTN does a better job of assigning more work to faster servers. When load increases, server performance becomes more variable: concurrency across queries increases query delay variability, and the Java virtual machine memory management operations (garbage collection, etc) affect the query runtimes. In such cases, assigning work only to the fastest servers is not necessarily best. That is why ROAR - which spreads its work more across servers -

outperforms PTN.

The main point of this comparison is that inherent variability in runtime speed tends to negate the effects of perfect scheduling: there is a fundamental tension between achieving the lowest query delays and coping with variability in runtimes. ROAR does a fair job of obtaining query delays under moderate load levels.

The shapes of the curves are similar for all three scenarios, yet the slope is gentler for the disk-bound experiment: there increasing p brings fewer benefits. Different users experience different read speeds on the same machine, depending on where the user file is placed on disk and how contiguous it is. Because of this variability, the frontend is fundamentally unable to estimate per user speeds from the average it maintains per ROAR server. The more servers are involved in a search, the more it is likely that one will be slow reading their part of the data.

Surprisingly, for the disk-bond experiment, the optimal p to achieve minimum delay depends on the system's load. As in the large scale experiment, increasing p to maximum does not guarantee minimum delay. This implies that achieving minimal delay is not possible with static p: adaptation is required.

7.10 Evaluation Summary

Our evaluation shows in a practical system that ROAR can indeed scale PPS to many objects (we ran as many as 5 million) and many users while providing low query delay. ROAR allows easy adaptation of p while the system is servicing queries providing a knob to optimize the system dynamically. We used this to automatically adapt p to reach a target delay, and measured the power benefits that can be had.

We cross-validated the analytical comparison to PTN, and found that in practice ROAR achieves similar query delays to PTN, while being able to seamlessly change p at runtime. The ROAR scheduling algorithm, although unoptimized, can support tens of queries per CPU core, and can easily be scaled up.

To evaluate at scale, we ran ROAR on 1,000 servers on Amazon. We found that even at such scale none of our centralized components became bottlenecks, and that changing p is an effective way to control query delay. We also found that high values of p can increase end-to-end delays because of TCP incast problems in certain network configurations. This increases our belief that the ability to seamlessly change p is a useful addition to all distributed rendezvous systems used in practice.

Chapter 8

Related Work

ROAR builds upon a large body of work in the distributed systems literature. ROAR is most related to distributed rendezvous search systems running in data centers where data is placed on nodes and queries routed without taking content into account. Existing systems in this category are reviewed in detail in Section 8.2. We provide background information on the alternative of using content to distribute work in Section 8.1

There has been a tremendous amount of work in providing peer to peer search solutions, which in turn has built upon work in structured overlays. ROAR builds upon ideas from this work which is reviewed in Section 8.3.

Content-based publish/subscribe systems are conceptually close to search, some researchers dubbing them as two sides of the same coin [BC92]. Section 8.4 provides an overview.

Large scale processing was initially researched in distributed databases targeting parallel execution of SQL queries. We conclude our survey of related work with a brief outlook of distributed databases in Section 8.5.

8.1 Content Based vs. Content Insensitive Distributed Search

Is Distributed Rendezvous the proper solution for search? Here we position distributed rendezvous against alternative solutions. Distributed Rendezvous is content insensitive: it does not use contents of the data or the queries for replica placement or query execution. The alternative is to use content when executing distributed queries.

Distributed Rendezvous has two advantages over content-sensitive solutions: simplicity and generality. The fact that it does not take into account content means that it is immune to skewed content distributions or variations of such; thus mechanisms to deal with these are not needed. Distributed Rendezvous is general in that it can support a wide variety of queries and data types: any algorithm taking as input data objects and answering yes/no can be used as a query. DR can easily employ randomization as it does not care about content. This is a big gain, as it allows good load balancing. All giant-scale services seem to employ randomization [Bre01].

On the downside, its content agnostic approach precludes content-based optimizations, such as indices on distributed tables, or smart clustering of web documents based on content. Indeed, much

research effort has gone to devising techniques to cluster web documents based on content (i.e. LSI based, keyword based), to cluster attribute values and support range queries (Mercury), etc.

In information retrieval, there are two main ways of partitioning a collection of documents amongst servers: keyword-based or document-based [MWZ06]. Document-based partitioning is similar in spirit to DR, as it does not take into account content when assigning documents to servers, or queries to servers. This solution has natural load balancing, and is easier to manage as each server computes its own index subset locally. However, if the collection is stored on disk, more seeks are required than in the keyword-based version.

Keyword-based partitioning uses keywords in documents and queries to guide document storage and query execution. It has fewer disk seeks but imbalance can indeed kill it. Moffat et al. have studied numerous techniques to improve load balancing and throughput of keyword-based partitioning, including replicating the highest volume keywords and balancing keywords across servers according to their frequency; what they have found is that for a deployment on a small number of servers, the throughput of the optimised keyword-based scheme is always a bit lower than that of document-based partitioning [MWZ06]. The main reason for this decrease are short term load imbalances amongst the servers. In short, a mechanism that seems better in theory is more complex in practice and achieves similar results to the simple document-based partitioning in practice.

Furthermore, content-based solutions suffer if content in data and queries is highly skewed, if content distribution varies, and do not work when content is unknown—as is the case with privacy preserving search.

8.2 Distributed Rendezvous Solutions

There are many proposed distributed rendezvous solutions in the literature [BDH03, FRA+05, TBF+04, TKLB07, GS04]; some target peer to peer deployments and some data centers, yet almost all offer a fixed trade-off between the partitioning and replication levels.

The Google cluster architecture [BDH03] (PTN) is the classical cluster-based solution, with a fixed r-p trade-off. We have analyzed it extensively throughout this thesis, finding that changing the p-r tradeoff is quite difficult. ROAR achieves delays similar delays to PTN's while being able to reconfigure the system on the fly, at runtime.

Another similar solution is the Load Balancing Matrix (LBM) [GS04]. LBM is the only solution we are aware of that allows changing r dynamically. LBM use the same cluster structure as PTN, but at a virtual level: the clusters are mapped onto a Distributed Hash Table. Server i from cluster j is mapped to the DHT server in charge of $hash(i, j)$. When repartitioning, LBM inherits all the problems of the Google approach. Furthermore, 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. Because of the virtual mapping LBM loses the nice load balancing properties of PTN. There is no easy way to fix this. One solution is that 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. Furthermore, the appealing simplicity of PTN - where all servers

in a cluster stored identical data - is lost, without any major gains.

BitZipper [TBF+04] is a distributed rendezvous solution which is also routed on top of a DHT and aimed at peer to peer deployment; here the tradeoff between replication and partitioning is fixed, as $p \sim r \sim \sqrt{n}$. BitZipper is optimised to minimize total throughput. In data centers bandwidth usage for search applications is not a big concern. If it were, ROAR can be used to control p and r such that they are proportional to $\sqrt{n}$, also minimizing bandwidth usage.

A few randomized solutions have also been proposed, and they are similar to the RAND algorithm we have described in Section 3: Ferreira et al. [FRA+05] use random walks for both object storing and for queries, while BubbleStorm [TKLB07] uses bubbles to speed up object storing and query execution. These algorithms are built for peer to peer systems and have great resilience to node churn (nodes coming and going) and failures. For instance, BubbleStorm [TKLB07] can withstand 50% node failures, without needing any centralised component. They offer probabilistic guarantees of finding objects, and these may not be good enough in data center like searches. Also, their operating costs are much higher (for instance with BubbleStorm $p \cdot r = 4n$), needing more hardware and more energy to do the same amount of work. For this reason, randomized solutions are not suitable to data center environments where failure rates are low.

Glacier [HMD05] is a distributed storage system that replicates objects to r equally-spaced servers on a Chord ring to improve availability. It would be easy to turn Glacier into distributed rendezvous: route each query to all the servers in a $1/r$ arc. However, to change the replication levels each server needs to record the servers in charge of the previous and next replica for every object, resulting in memory and bandwidth costs and creating consistency issues when servers fail. ROAR deliberately chooses the dual approach to eliminate these problems: queries (transient in nature) are routed to where the objects would be stored; object-consistency issues are reduced to synchronising with neighbours.

Beehive [RS04] replicates objects to achieve one hop lookups, assuming object popularity is Zipf distributed. Beehive's replica placement algorithm stores replicas on servers at Hamming distance 1 in the ID space, and only allows values for r that are a power of 2. It appears easy to use Beehive's replica placement strategy for distributed rendezvous, but the complexity of object replication combined with restrictions on values of r limit possible benefits.

Chain replication [vRS04] is similar in spirit to ROAR's object placement strategy. Van Rennesse et al. show that to achieve consistency and high throughput, object updates should be serialized at the home server and queries should be executed by the active server. This works when individual objects are accessed; in distributed rendezvous all objects must be accessed so using this strategy is infeasible, as the query must be broadcast to all the servers.

8.3 Structured Overlays and Peer to Peer Search

Structured overlays like Chord [SMK+01], Pastry [RD01] or CAN [RFH+01] organise a large set of servers into a network structure that has low degree and low diameter. Their biggest appeal is complete decentralisation: nodes are completely self organising, there are no centralised components.

Chord [SMK+01] assigns each server a random identifier in a 160 bit circular space. Each server is

in charge of the range between himself and its predecessor. It maintains links to a few of its predecessor and successor nodes, and also $\log n$ "finger" pointers to nodes further out on the ring. Key lookup is the basic operation supported by Chord: starting from an initiator node, the request is routed closer to the desired ID by using the neighbour or finger pointers; on average $\log n$ steps are required. Objects are replicated in Chord on their home node and a few successors.

Pastry [RD01] has a similar structure and similar properties to Chord. CAN uses a d dimensional torus, having constant node degree and larger diameter than Chord.

ROAR borrows the idea of ring from Chord. ROAR replication has a subtle but important difference from Chord: ROAR objects are replicated on a fixed ID range rather than on a fixed number of servers. This allows us to decouple server and object locations, a crucial property for practical distributed rendezvous. Further, ROAR has centralised server membership, its servers do not maintain finger pointers, and it does not use logarithmic lookup to run queries. All these are possible in data centers, and allow ROAR to obtain low query delays that couldn't otherwise be obtained.

A number of systems have been recently deployed in data-centers to allow key-value stores or database-like functionality, including Dynamo [DHJ+07] and Cassandra [LM10]. Dynamo is essentially a one hop DHT built on top of Chord that favours availability and partition tolerance over consistency; it uses vector clocks to detect conflicts and defers the conflict resolution to the application. Cassandra is similar to Dynamo, but offers richer functionality than the key-value store. Cassandra supports database-like row and column-based operations, but does not explicitly support SQL. Both systems use data replication and partitioning as building blocks, as ROAR does. Cassandra, in particular, allows quorum reads and writes with configurable ratios; the quorum uses the same intersection property that ROAR uses to achieve its functionality. Cassandra basic storage mechanism is borrowed from Chord, and it has many problems when used in a distributed rendezvous context (these were discussed in detail in Chapter 3.3).

Chord, CAN and Pastry only offer basic primitives to store and retrieved named items; hence they are collectively called distributed hash tables (DHT). This basic functionality does not support keyword search, however much research has gone into executing queries on DHTs, including keyword search [RV03, TXD03, TD04] and range queries [BAS04].

Straightforward partitioning of documents based on keywords is proposed by Reynolds at al [RV03]. Techniques are devised to minimise the amount of communication between servers, when intersecting document lists. Load balancing is ignored in this work, which questions its scalability. ESearch [TD04] goes a step further and only uses the top 25 keywords in each document for indexing, observing that this suffices usually, and by replicating full document information at these nodes. Load balancing is performed for document storage by using DHT specific mechanisms. In effect, partitioning replicates a keyword on multiple nodes, and therefore query load balancing is not dealt with directly. We have seen in our experiments that load balancing is paramount to achieving high throughput.

In an attempt to even out the balancing of inverted indexes to nodes, Liu et al. propose to remove inverted lists pertaining to highly used words and distribute this information to the other nodes [LL04].

Conjunctive queries containing popular terms will be answered with documents that contain the less popular terms. Information about the fusion dictionary is replicated to all the nodes in the system, and so are the lists of files that contain only these words. When multiple keywords are removed, their lists are aggregated and a synthetic keyword is used to store documents that contain those two keywords. This in fact removes the initial mappings and introduces correlated mappings for frequently occurring pairs; based on our analysis in [Rai06], we show that the number of routing hops is not reduced drastically and therefore the impact of this heuristic is not substantial.

PSearch [TXD03] uses document and query content to store documents and route queries on top of CAN. Instead of using the actual terms, PSearch uses Latent Semantic Indexing to map all the documents and queries onto a multidimensional semantic space, which is then mapped onto a CAN with a lower number of dimensions. The authors propose techniques to reduce the number of needed dimensions by using a rolling index and replicating each document a few times on the overlay. They also propose techniques to balance the document load on the servers. Despite these techniques, load is still skewed: in the authors' experiments, 35% of the servers store 70% of the indices with all the techniques enabled. These results outline once again the fundamental difficulty of properly balancing documents with content-sensitive placement.

In general, directly using the peer to peer search techniques for data center deployments seems wrong. An interesting discussion about the feasibility of scaling peer-to-peer search for web search is provided in [LLH+03], and it is shown that currently the resource usage would be one order of magnitude higher than the resources available. Google is clear evidence that in data centers web search is feasible. To support it, however, techniques must be designed specifically to take advantage of the properties of the data centers that differentiate them from peer to peer systems: zero churn, single administrative domain, high bandwidth and low failure rates. While ideas from ROAR are more general and could be applied to peer-to-peer search, ROAR has also been designed for use in data centers.

8.4 Content-Based Publish/Subscribe Systems

Content-based publish/subscribe (CBPS) is an interaction model where interests of subscribers, expressed as predicates over the desired attributes in notifications, are stored in a content based matching infrastructure. Publishers push notifications into the infrastructure, and the notifications are typically attribute name-value pairs. The infrastructure's task is to decide which notifications should go to which subscribers in an online manner. Research has focused on how to distribute this intermediary to scale interaction to a large number of users. Content-based publish subscribe systems are conceptually close to search: if stored subscriptions are replaced by documents, and notifications by queries, we have a distributed search system. Hence, solutions from CBPS can in principle be applied to distributed search.

Existing architectures for content-based publish/subscribe can be divided roughly in two categories: fixed topology architectures (the traditional approach to pub/sub, including Siena [CRW01], Gryphon [gry99], their optimisations and variants) and DHT-based architectures, including Mercury [BAS04], Homed [CPP04] and so on. In the first category, event routing is intertwined with content-based matching, with the noticeable exceptions of Medym [CS05] and EDN [CDNF01]. The second class of applica-

tions usually focuses on resilience to node churn and fault tolerance (inherited from DHTs), attempting to minimize the number of routing hops for matching and putting in the same time less focus on event routing.

Most architectures assumes fixed topologies (typically trees or directed acyclic graphs) that are created by the users (Siena [CRW01], Gryphon [gry99], Medym [CS05]). The algorithms are generally conceived for fault-free operation. Assuming that the connections between nodes are (manually) established based on network proximity, fixed topology architectures achieve minimal network delay for delivering messages to each subscriber. Their biggest advantage is low bandwidth usage: for each notification, a multicast tree is built which replicates the notification as late as possible. Perhaps the biggest drawback is their worst-case behavior: the number of routing hops a notification traverses grows linearly with the number of nodes in the system and every subscription gets replicated to every node (for both Siena [CRW01] and Gryphon [gry99]). Also, application level content-matching is usually performed at each hop, incurring significant delays for notifications crossing a large number of hops.

If these systems were used for distributed search, a lot of the optimizations they embed would become meaningless. For instance, when running a query all the results need to be returned to the front-end: there is no need to create a multicast tree to subscribers as in CBPS. Further, CBPS event routing is sequential, whereas distributed search systems such as Google's or ROAR send a query to many servers in parallel to reduce query delay. To conclude, these solutions are not applicable in practice to distributed search.

8.5 Relational Databases

Speeding up queries has always been a goal of database systems, and parallel databases [Cor83, CABK88, DGG+86, LDH+89, SAL+96, TD03, KW94] provided a shift in this direction. The trend was to move away from mainframe computers using either shared-disk or shared-memory computer architectures towards shared-nothing machines [DG92, Sto86].

To parallelise execution of queries the most effective technique is to horizontally partition the input table across many nodes [DG92], such that each node can execute the query in parallel on its subset of the data; the results are then merged. The biggest dangers to scaling up query processing to larger datasets and speeding up queries were identified to be skew (load imbalance), startup costs and self interference from different parts of the same query [CABK88, GD90, DG92].

Partitioning can be round robin (suited for sequential scans), hash-based (suited for sequential scans and associative access) and range partitioning [DG92]. Range partitioning, used in Arbre, Bubba, Gamma, Oracle and Tandem, is similar in concept to content-based placement of data and queries, and can suffer from load imbalance. Adjusting the range sizes can counter this effect, as in [CABK88] The problem with using range partitioning for full text searches is that there are too many dimensions to partition (e.g. see the latent semantic space, as in pSearch [TXD03]). As such, hash based-partitioning (used by Arbre, Bubba, Gamma and Terradata) is preferrable for search. This is also what ROAR uses.

How much should a table be partitioned? Partitioning too much might have adverse effects, increasing query delay and decreasing system throughput [CABK88, GD90]. This is the same observation we

made both qualitatively and quantitatively in this thesis. Traditional parallel databases have fixed partitioning strategies, whereas ROAR provides a means to dynamically trade-off query overhead vs. query delay.

In general, research in distributed databases aims to optimise execution of powerful relational queries in a distributed setting. 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 optimise 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 [GD93].

Distributed Computation. There are many other algorithms for distributing computation among many machines, such as [BTAD+04, ADAT+99, DG04, YIF+08]. Google's MapReduce [DG04] offers a simplified, functional programming model that hides parallelisation from the programmer. ROAR offers a weaker programming abstraction, equivalent to the "map" operation, but differs in its handling of data objects: while Map Reduce 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. The major difference between ROAR and MapReduce is their intended use: MapReduce optimises execution of large jobs on huge datasets that take from seconds to hours to execute, while ROAR is aimed at running sub-second queries against smaller amounts of data. By not copying data for every query allows ROAR to save bandwidth and obtain smaller delays.

Chapter 9

Conclusions

The performance of distributed rendezvous systems such as web search engines is heavily influenced by the partitioning level p, which controls how an ensemble of servers handle queries and store data. This parameter is the primary control that determines search latency, and so has a huge impact on the usability of distributed search systems. We have found that the query delay variation as p increases is not necessarily monotonically decreasing, as it depends on the load on the servers and on the particularities of the search application. Further, higher values of p increase the fixed costs associated to queries be they hard drive seeks, OS related overheads, or network bandwidth costs. This dependency matters: on our small cluster, running with $p = 50$ instead of $p = 5$ wastes energy costing 50$ per server per year. In the three year lifetime of a 1,000$ server savings could be up to 150$. While not astounding, these savings are worthwhile.

Fixed costs associated to queries naturally push p down as long as delay targets are met. We have further shown that object updates push r to be small, otherwise server throughput is affected. To be efficient, a distributed rendezvous system needs to run at its optimal operating point, where $pr = n$.

Despite this and the fact that p should be continuously adapted according to the system's load in order to achieve optimal performance, search engines such as Google rely on the simple PTN algorithm that does not allow for dynamic reconfiguration of p. PTN's simple structure fundamentally prevents it from reconfiguring easily, as it loads nodes asymmetrically during changes, and reduces the capacity of the system while change is taking place.

9.1 Contributions

The premise of this thesis is that allowing seamless reconfiguration brings alive another dimension on which the distributed rendezvous system can be optimised continuously to track the load it is serving.

We have introduced ROAR, a novel distributed rendezvous algorithm that allows on-the-fly reconfiguration 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. ROAR uses rings organize servers, and uses fixed replication ranges to store documents on servers. This allows it to decouple query routing and replica placement from server density on the ring, overcoming the main

problems with the SW algorithm.

We have created an analytical model to study the properties of ROAR and compare it to PTN. We have explored the parameter space comparing ROAR, PTN and SW along multiple dimensions including query delay, availability, and the ability to reconfigure.

One major challenge in ROAR was to provide comparable query delay to PTN. We achieved this through three main techniques:

- **Server range balancing** assigns larger ranges to more powerful servers, helping overall load balancing and reducing query delays.
- **Multiple rings** provide the power of two choices for query execution, breaking away from the r choices available to the SW algorithm.
- **Range adjustment** is a local heuristic that reduces the delay of the sub-query running on the most loaded server.

These three mechanisms work harmoniously together to reduce ROAR query delays. Through simulations we compared the query delays of PTN and ROAR, finding that ROAR outperforms PTN when server speeds are fluctuating and can't be perfectly predicted, while PTN behaves better when server speeds are more predictable. ROAR can reconfigure seamlessly as it places uniform, minimal bandwidth load on all servers during the change. In contrast PTN asymmetrically loads servers and takes orders of magnitude more time to reconfigure. Further simulations have shown that ROAR and PTN both give high availability,

To test ROAR in practice we implemented a privacy-preserving search application that used ROAR as its underlying algorithm, and ran experiments on a 50-server dedicated testbed and on a 1000-server configuration using Amazon's EC2.

Our practical experiments 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. 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 minimising other costs. All these results give us confidence that ROAR is a practical alternative to PTN, offering an important knob to optimise system behaviour - p - a knob that does not exists in systems today. Distributed rendezvous systems can make more efficient use of resources by continuously adapting to load and environment changes.

We also performed a head-to-head comparison between ROAR and PTN, measuring query delays. The results cross-validated our simulations, showing that ROAR delays are better than those of PTN when load is moderate or high, while PTN is better when load is low.

The other major contribution of this work is Privacy Preserving Search. PPS addresses a major privacy concern raised by the "online" convergence of user data by supporting search when user data are stored encrypted. Simulation studies indicate that, at least for mobile devices, PPS should consume a lot less bandwidth than and is thus preferable to the naive solution of encrypting the search index.

To support PPS we have proposed novel cryptographic constructions to match numeric attributes, and built a search system that allows moderately complex privacy preserving searches, including keyword, modification date and file size searches. Our experimental analysis of PPS on a single machine outlines the need for parallelism: a single query running against 1 million metadata takes well over 4s.

We parallelised PPS with ROAR and achieved sub-second query delays for much larger datasets (5 million files). As user queries are relatively infrequent, a set of machines can be used to server many clients. We believe it should be possible to build an economically viable PPS system; however this is not the purpose of this -thesis.

9.2 Future Work

In the future, we hope to test ROAR more on large clusters with thousands of nodes, and explore different optimisation criteria. Most interestingly, it would be great to see if ROAR can be directly applied to web search in practice.

ROAR opens an interesting research question: what are the fundamental techniques we can use to create efficient distributed algorithms? Energy efficient computing has been mostly looking at individual components such as CPU, disk or whole servers to try and make their energy usage load adaptive. Our work shows that different values of p always change the total amount of work done by the system, and that controlling p opens the door for further energy savings. In its quest for efficiency, ROAR makes the distributed algorithm rather than the individual components the focus of optimisation. This new focus seems promising.

Bibliography

[ADAT+99] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster i/o with river: making the fast case common. In *Proc. Workshop on I/O in parallel and distributed systems*, 1999.

[AFLV08] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proc. SIGCOMM*, 2008.

[AGM+10] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proc. Sigcomm 2010*, volume 40, 2010.

[Ama] Amazon. Elastic compute cloud.

[BAS04] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 34(4), 2004.

[BC92] Nicholas J. Belkin and W. Bruce Croft. Information filtering and information retrieval: two sides of the same coin? *Commun. ACM*, 35(12):29–38, 1992.

[BDH03] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *Micro, IEEE*, 23, 2003.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 1970.

[Bre01] Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[BTAD+04] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control a batch-aware distributed file system. In *NSDI*, 2004.

[CABK88] George Copeland, William Alexander, Ellen Boughter, and Tom Keller. Data placement in bubba. *SIGMOD Rec.*, 17(3):99–108, 1988.

[CDNF01] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. Softw. Eng.*, 27(9):827–850, 2001.

[CFSS05] Chris Chambers, Wu-chang Feng, Sambit Sahu, and Debanjan Saha. Measurement-based characterization of a collection of on-line games. In *IMC 2005: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[CGKO06] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 79–88, New York, NY, USA, 2006. ACM.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE Symposium on Foundations of Computer Science, FOCS*, pages 41–50, 1995.

[CGL+09] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *WREN 2009: Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 73–82, New York, NY, USA, 2009. ACM.

[CHL+08] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI 2008: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 337–350, Berkeley, CA, USA, 2008. USENIX Association.

[CM05a] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Proc. ACNS*, 2005.

[CM05b] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS*, 2005.

[Cor83] Teradata Corp. Teradata: Dbc1012 data base computer concepts and facilities. Document No. C02-0001-00, 1983.

[CPP04] Y. Choi, K Park, and D. Park. Homed: A peer-to-peer overlay architecture for large-scale content-based publish/subscribe systems. In *3rd International Workshop on Distributed Event-Based Systems (DEBS'04)*, 2004.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[CS05] Fengyun Cao and Jaswinder Singh. Medym: Match-early with dynamic multicast for content-based publish-subscribe networks. pages 292–313. 2005.

[CW03] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *Proceedings of ACM SIGCOMM*, pages 163–174, Karlsruhe, Germany, August 2003.

[Dea] Jeffrey Dean. Personal Communication. Google.

[DG92] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[DGG+86] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 228–237, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. 2002.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE Trans. Netw.*, 1(4):397–413, 1993.

[FKN94] Uri Feige, Joe Killian, and Moni Naor. A minimal model for secure computation (extended abstract). In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 554–563, New York, NY, USA, 1994. ACM.

[FRA+05] Ronaldo A. Ferreira, Murali Krishna Ramanathan, Asad Awan, Ananth Grama, and Suresh Jagannathan. Search with probabilistic guarantees in unstructured peer-to-peer networks. In *Proc. P2P*, 2005.

[GD90] Shahram Ghandeharizadeh and David J. DeWitt. A multiuser performance analysis of alternative declustering strategies. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 466–475, Washington, DC, USA, 1990. IEEE Computer Society.

[GD93] G. Graefe and D. L. Davison. Encapsulation of parallelism and architecture-independence in extensible database query execution. *IEEE Trans. Softw. Eng.*, 19(8), 1993.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP 2003: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[GHMP09] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, 2009.

[GLL+09] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. In *Proc. SIGCOMM*, 2009.

[Goh03a] Eu-Jin Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003.

[Goh03b] Eu-Jin Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. `http://eprint.iacr.org/2003/216/`.

[Gol01] Oded Goldreich. *Foundations of Cryptography*, volume Basic Tools. Cambridge University Press, 2001.

[Gre09] Albert Greenberg el al. VL2: a scalable and flexible data center network. In *Proc. ACM Sigcomm*, 2009.

[gry99] An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 262, Washington, DC, USA, 1999. IEEE Computer Society.

[GS04] Jun Gao and Peter Steenkiste. Design and evaluation of a distributed scalable content discovery system. *IEEE Journal on Selected Areas in Communications*, 22, January 2004.

[HGSW10] Daniel Halperin, Ben Greensteiny, Anmol Shethy, and David Wetherall. Demystifying 802.11n power consumption. In *Proc. HotPower*, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.

[HMD05] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. NSDI*, 2005.

[HMT04] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. A privacy-preserving index for range queries. In *Proceedings of VLDB - Conference on Very Large Databases*, 2004.

[IK97] Yuval Ishai and Eyal Kushilevitz. Private simultaneous messages protocols with applications. In *ISTCS '97: Proceedings of the Fifth Israel Symposium on the Theory of Computing Systems (ISTCS '97)*, page 174, Washington, DC, USA, 1997. IEEE Computer Society.

[KHF06] Eddie Kohler, Mark Handley, and Sally Floyd. Designing dccp: congestion control without reliability. *ACM SIGCOMM*, 2006.

[KKG+10] Michael Kounavis, Xiaozhu Kang, Ken Grewal, Mathew Eszenyi, Shay Gueron, and David Durham. Encrypting the internet. In *Proc. Sigcomm 2010*, 2010.

[KW94] Brigitte Kröll and Peter Widmayer. Distributing a search tree among a growing number of processors. *SIGMOD Rec.*, 23(2), 1994.

[Lam01] Leslie Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, December 2001.

[LDH+89] R. Lorie, J. Daudenarde, G. Hallmark, J. Stamos, and H. Young. Adding intra-transaction parallelism to an existing dbms: Early experience. *IEEE Data Engineering Newsletter*, 12(1), 1989.

[LL04] Lintao Liu and Kang-Won Lee. Keyword fusion to support efficient keyword-based search in peer-to-peer file sharing. In *CCGRID 2004: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, pages 269–276, Washington, DC, USA, 2004. IEEE Computer Society.

[LLH+03] Jinyang Li, Boon Loo, Joseph Hellerstein, M. Kaashoek, David Karger, and Robert Morris. On the feasibility of peer-to-peer web indexing and search. In *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[Mit01] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12:1094–1104, 2001.

[MWZ06] Alistair Moffat, William Webber, and Justin Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR 2006: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 348–355, New York, NY, USA, 2006. ACM.

[NDR08] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):1–23, 2008.

[NPI+08] Sergiu Nedevschi, Lucian Popa, Gianluca Iannaccone, Sylvia Ratnasamy, and David Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *NSDI 2008: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 323–336, Berkeley, CA, USA, 2008. USENIX Association.

[oST95] National Institute of Standards and Technology. Secure hash standard, 1995.

[Rai06] Costin Raiciu. Phd transfer report: On distributed online filtering. UCL, 2006.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proc. SIGCOMM*, 2001.

[RR06] Costin Raiciu and David S. Rosenblum. Enabling confidentiality in content-based publish/subscribe infrastructures. In *Proc. Securecomm*, 2006.

[RRH07] Costin Raiciu, David S. Rosenblum, and Mark Handley. Distributed online filtering. In *Poster Session: ACM Sigcomm*, 2007.

[RS04] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proc. NSDI*, 2004.

[RV03] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Middleware 2003: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 21–40, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[SAL+96] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: a wide-area distributed database system. *The VLDB Journal*, 5(1), 1996.

[SG07] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, page 1, Berkeley, CA, USA, 2007. USENIX Association.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. SIGCOMM*, 2001.

[SNR+10] Aaron Schulman, Vishnu Navda, Ramachandran Ramjee, Neil Spring, Pralhad Deshpande, Calvin Grunewald, Kamal Jain, and Venkata N. Padmanabhan. Bartendr: a practical approach to energy-aware cellular data scheduling. In *Proc. Mobicom*, pages 85–96, New York, NY, USA, 2010. ACM.

[Sto86] Michael Stonebraker. The case for shared nothing. *Database Engineering*, 9:4–9, 1986.

[SWP00] Dawn Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2000.

[TBF+04] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Jussi Kangasharju, and Alejandro Buchmann. Bit zipper Rendezvous—Optimal data placement for general P2P queries. In *Proc. EDBT Workshop on Peer-to-Peer Computing and DataBases*, 2004.

[TD03] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *Proc. VLDB*, 2003.

[TD04] Chunqiang Tang and Sandhya Dwarkadas. Hybrid global-local indexing for effcient peer-to-peer information retrieval. In *NSDI 2004: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[Tec10] Ars Technica. Ftc reminds us that storing data in the cloud has drawbacks, January 2010.

[TKLB07] Wesley W. Terpstra, Jussi Kangasharju, Christof Leng, and Alejandro P. Buchmann. Bubblestorm: resilient, probabilistic, and exhaustive peer-to-peer search. In *Proc. SIGCOMM*, 2007.

[TXD03] Chunqiang Tang, Zhichen Xu, and Sandhya Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proc. Sigcomm*, 2003.

[VPS+09] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *ACM SIGCOMM*, 2009.

[vRS04] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. OSDI*, 2004.

[WAB+06] Christian Wallenta, Mohamed Ahmed, Ian Brown, Steven Hailes, and Felipe Huici. Analysing and modelling traffic of systems with highly dynamic user generated content. University of Oxford Research Note RN/08/10, 2006.

[web09] The size of the world wide web. http://www.worldwidewebsize.com/, November 2009.

[Yao86] Andrew C. Yao. How to generate and exchange secrets. In *Proceedings of the IEEE Symposium of Foundations of Computer Science, FOCS*, 1986.

[YGN06] Haifeng Yu, Phillip B. Gibbons, and Suman Nath. Availability of multi-object operations. In *Proc. NSDI*, 2006.

[YIF+08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.