UNIVERSITEIT TWENTE.

Sharding Spotify Search

by

Emiel Mols

A thesis submitted in partial fulfillment for the
degree of Master of Science

in the

Faculty of Electrical Engineering, Mathematics and Computer Science
Department of Computer Science

August 2012

"We wanted flying cars, instead we got 140 characters."

Peter Thiel, from the Founders Fund manifesto

UNIVERSITEIT TWENTE.

Abstract

Faculty of Electrical Engineering, Mathematics and Computer Science
Department of Computer Science

Master of Science

by Emiel Mols

We rationalise and describe a prototype implementation of a term sharded full text search architecture. The system's requirements are based on the use case of searching for music in the Spotify catalogue. We benchmark the system using non-synthethic data gathered from Spotify's infrastructure.

Preface

This thesis finds its genesis in a healthy quantity of *happenstance*. Perhaps a combination of me having a fascination for the country of Sweden, inspiration for a technically interesting side project [1], and then this endevour to change the music marketplace called Spotify existing in the first place.

And these things only relate to the likeliness of this research project getting off the ground. During the last ten months, there have been more than a few moments of doubt and self reservation. Ultimately, though, those reservations never overtook my appreciation for academia.

Many of my fellow students seem to regard academia as another hurdle between them and a bigshot carreer, university as a tool to bump their salary or a graduation project at an established company as some shiny star on their resume.

I am convinced these are the smallest of merits. Academia enabled me to question the most fundamental of concepts and helped me to understand the world a little better. That's what made it worthwhile. [2]

[1] A personal project got some press and allowed me to of do this thesis project at Spotify; http://thenextweb.com/insider/2011/05/20/how-to-give-spotify-no-choice-but-to-hire-you/.

[2] The articulate reader might have noticed that the inspirational quote on the previous page was put forth by a man who is well known for stimulating people to drop out of university. I don't agree with mister Thiel in this regard, but, well, it's just a great quote.

Contents

Chapter 1

Introduction

1.1 Full text search systems

A full text search system (*fts-system*) enables information retrieval by responding to a user's *query string* with an ordered set of documents. The role of the system is to identify the relevant documents for each query and minimize the time a user of the system would need to access the document she or he is looking for.

Well known fts-systems include Google web search [Brin and Page 1998], Facebook friend search [Lampe et al. 2006] and the computerized catalog system at your local library.

A principal property of such information retrieval systems is that the amount of documents contained in it (or, *corpus*) is often significant, presenting algorithmic and architectural challenges related to result latency, result relevancy and user scalability.

Requirements of fts-systems are diverse: for instance, where a web search engine could order results by a single metric such as percieved global relevance of the document, a product search system could allow ordering by arbitrary fields (e.g. price, release year, delivery time). Also, even though almost all systems are built to support a corpus of significant size, the order of magnitude of documents in a modern web search engine differs from the amount of news articles of an online paper. In spite of this, fundamental techniques between such diverse systems are often similar, or even identical.

1.1.1 Index construction

The most often used indexing technique in fts-systems is *inverted list* construction [Manning et al. 2008]: input documents are tokenized into words, or *terms*. These are then

used to construct a mapping from terms to an ordered list of document identifiers. Using this mapping and a *query*, the result space can be obtained by *merge joining* the document identifiers of the lists associated with the query terms.

An in-depth overview of the *inverted list* approach and associated algorithms is given in section 2.1.

1.1.2 Distribution

When considering the distribution of an inverted full text index over multiple hosts, the distinction can be made between *document based distribution* and *term based distribution* sharding [Moffat et al. 2007] (sometimes called *local index construction* and *global index construction*, respectively [Ribeiro-Neto and Barbosa 1998]). Both paradigms employ a number of subsystems to host the partitions (or, *shards*), and additional controlling functionality, such as a *query broker* to execute queries or an *indexer* to add data to the index.

Document-wise sharding entails distributing the data across different hosts as collections of input documents: each shard maintains a complete vocabulary and inverted full text index for the assigned documents.

In a term distributed architecture the inverted index is sharded by term: a shard hosts a set of complete inverse lists; document identifiers in these lists can be the same at multiple shards.

Document distribution, compared to term distribution, has the following advantageous characteristics:

- **Simplicity**. Document distribution is a simple extension of a non-distributed index; some relatively simple merger logic is required on the query broker [Banon 2012].
- **Latency and parallizability**. A broker can query the different shards in parallel, and for a reasonable amount of shards the query latency, excluding network overhead, will be similar to that of a non-distributed index (for a large number of shards, latency variance can increase significantly [Dean 2010]).
- **Indexing is well distributed**. The operation of amending a document to the index is handled by a single shard.

- **Balanced shards**. Because there is no correlation between the nature of the sharded data and the shard itself (assuming a large amount of similarly sized documents), both query and indexing load between the shards is uniform.
- **Predictable network load**. Only result sets - predictable in size - will be transferred over the network.
- **Graceful degradation**. When a shard (and its replicas - copies holding the same data - if applicable) are unavailable, the merger could still choose to merge and return data from the other shards.

Term distribution performs generally better on the following criteria:

- **Accessed shards bound by term in query**. For term distribed systems, the amount of shards touched when executing a query is bound by the amount of terms in the query; for document distributed systems (a fixed ratio of) all shards need to be accessed - this figure relates to the corpus size. As a consequence, term distribution allows for much more granular sharding without negatively influencing query performance. Also, best-case query latency (namely for single term queries) is better when the shard count is more than one.
- **Fewer seeks**. A document distributed system consists, in effect, of parallel index segments that all need to be accessed for each query: subsequently, the cumulative number of seeks per query is lower for term distributed systems [Moffat et al. 2007]. This is particularly influential on performance if the data structures are saved on hard disk media.
- **Reusability as distributed cache**. Because the data is inherently distributed by term (as implied by the name), a term distributed system can distribute its *query cache* (caching results of complete user queries) using the same term-to-shard mapping. For instance, when executing `robbie williams` would imply the inverted list of `robbie` would be accessed first, this shard could also hold the cache for the complete query. For document distributed systems, the cache size is often limited to memory capacity of the query broker and does not scale with the amount of hosts.

These two approaches have been compared extensively, but the conclusions of individual works differ, based on the exact performance criteria, datasets used, etc. [Ribeiro-Neto and Barbosa 1998] [Moffat et al. 2007] [Jeong and Omiecinski 1995] [Tomasic and Garcia-Molina 1993]. Practical implementations such as Google [Brin and Page 1998]

and Yahoo [Baeza-Yates and Cambazoglu 2008] use document distribution. A possible reason for this is that the non-uniform distribution of inverse list sizes makes term based distribution problematic [Baeza-Yates and Cambazoglu 2008].

Existing distributed fts products also favor document distribution (e.g. ElasticSearch [Banon 2012], SOLR [Smiley and Pugh 2009] and IndexTank [Perez 2012]), and are often implemented by reusing an existing single-host system (e.g. Lucene [Hatcher and Gospodnetic 2004]).

1.2 Spotify

Spotify is a music streaming service, offering more than 15 million (active) users access to a library of more than 15 million music tracks [Kreitz and Niemela 2010] [Spotify 2012]. Distributing the music data is done through a combination of a Spotify-operated backend infrastructure and a peer-to-peer network, in which most non-mobile clients participate. The peer-to-peer network is solely used for offloading track data distribution. Metadata retrieval, search, and other services are handled by the Spotify backend.

The Spotify search backend service is implemented around Lucene and scales through replication of the indexes across several identical search servers. Separate Lucene indexes are maintained for artists, albums, tracks and public playlists and these are accessed in parellel for query requests. The cumulative corpus size of the indexes is around 30m. Indexes are kept in-memory, and currently take up around 7 gigabytes.

For the current dataset, the search service in its current incarnation performs satisfactorily, while leaving decent headroom for scaling in the near future. However, changing requirements could make the the index outgrow the availability of main memory by orders of magnitude, for instance if:

- a (much) larger corpus would need to be supported;
- more metadata might need to be added for the documents in the system, e.g. lyrics associated with songs, more extensive genre classification or biographical text with artists.

In chapter 3, we offer analyses on user behavior of the Spotify search system. The data gathered underlines arguments that support the case for term distribution:

- **Few terms in each query.** Queries entered into the system consist of relatively few terms - very often only one or two.

- **Few updates.** Changes to the index are rare in comparison to the query volume: hundreds of millions of queries are handled on a monthly basis, while the amount of updates over the same period is in the order of magnitude of 100.000.
- **High cache sensitivity.** A lot of queries, or parts thereof, are repetitious. Reusing the term distributed system as distributed cache might yield performance benefits. In combination with *query pipelining* (see section 2.5), we might even use such a cache to partially evaluate queries, e.g. use a cached result for the query `hans zimmer` to boost evaluation performance of queries such as `hans zimmer dark knight`.

1.3 Goal

The main goal of designing a new search architecture is to overcome the limits of the scalability of Spotify's current system through design and implementation of a new, *term distributed* search system.

By designing such a new architecture from the ground up, we can choose the specific data manipulation and distribution primitives tailored to the data domain and use case of Spotify.

The previous section outlines our main argument supporting the feasibility of such an architecture for the use case at hand. Additionally, we feel that the lack of proper implementations of term distributed systems thus far is specifically something academic research could tackle; a successful prototype could encourage the development - or even be the starting point - of a more full-fledged and usable system.

The primary goal for this research is therefore to present a functioning prototype of a term distributed fts-system. We compare its performance for Spotify's use case to the current backend service in use and a distributed setup of ElasticSearch - an open source distributed fts-system that employs document distribution [Banon 2012].

1.4 Contributions

We make the following contributions:

- Insight into user behavior in the Spotify search system, and, by generalization, user behavior when searching for music.

- The design and implementation of a prototype term distributed fts-system. Specifically, we developed three novel approaches to tackle existing problems that plague implementations of term distributed systems:
  - granular sharding of inverted lists that allow for parts of large inverted lists to be distributed among different shards, providing highly distributed read workloads;
  - *index segmentation* through a *forked* read process, where we use *copy-on-write virtual memory* semantics, allowing agressive read optimizations (and resulting in eventual read/write consistency of the index);
  - complex query pipelining, allowing arbitrarily complex boolean query expressions to be executed in a *query pipeline*.
- Determine the value of a term distributed search architecture over a document distributed one in the data domain of Spotify.

1.5 Contents

Chapter 2 will discuss a collection of engineering techniques foundamental to our prototype system. In chapter 3, we analyse Spotify's document collection and user behavior of the search system. Both chapters make arguments that influence the main design decisions of our new system, which we present in chapter 4. We then benchmark our new system through a series of - individual and comparitive - performance benchmarks in chapter 5. Finally, we validate our hypotheses around the system in chapter 6; for failed expectiations, we present directions for future research that might mitigate encountered problems.

Chapter 2

Database techniques

In this chapter we discuss a collection of engineering techniques used in various existing database applications and have relevancy to our engineering efforts for *swarm*.

2.1 Inverted index construction

An *inverted index*, constructed in fts-systems to prevent a complete *corpus scan* for each query, maps terms to an ordered list of documents that contain this term. The indexing step still requires scanning the entire corpus. A *single term* query subsequently consists of a simple index lookup.

Fundamentally, an inverted index consists of the *vocabulary* and the *postings list* [Manning et al. 2008].

- The *vocabulary* is a data structure that maps the terms to the associated *postings lists*. Implementations often use a B+-tree, trie or hash map.
- The *postings list* is an ordered list of the document identifiers that the term occurs in. For each document in a posting list, contextual data might be amended (e.g. a *hit list*, a list of offsets of the specific term in the document). Implementations often use unrolled linked list or skip lists.

The explicit distinction between these two structures is done because in many full text systems the vocabulary, orders of magnitude smaller in size, is suitable for in-memory operation, while the postings list may reside on a slower and larger storage medium, such as a hard disk [Manning et al. 2008].

In principle, though, postings lists can be integrated directly in the vocabulary's data structure, preventing a level of indirection and creating a *clustered* index.

On index generation, certain *stop words* can be ignored to prevent large lists in the inverted indexed (e.g. `in` or `and`), and *stemming rules* can be used to map different inflections of a word to a single term or term id (e.g. `dancing` and `dance`); the same stemming rules should be applied to terms in any query. Stop word filters or stemming rules are dependent on the locale of the indexed data, and most systems require them to be configured manually (e.g. Lucene's default installation is configured for the English locale [Hatcher and Gospodnetic 2004]).

The query model of a single term can be extended in various ways. Common methods described in literature and seen in popular full text retrieval systems are [Manning et al. 2008]:

- **Disjunction** (`abba OR queen`) involves a join strategy to find the union of two posting lists. A *sorted merge join* is applicable.
- **Conjunction** (`queen AND bicycle`, or implicitly `queen bicycle`) is similar, yet should yield the intersection of the relevant lists. Again a sorted merge join is often applicable, but there are exceptions. For instance, in the case of a significant difference in the amount of documents that match each term (e.g. `the cranberries`, where `the`'s cardinality is probably orders of magnitude higher than `cranberries`), a *zip-join* is probably more effective: for every item in the smaller list, a binary search is performed in the larger list.
- **Phrasing** (`"hotel california"`) would require appending the index with the positional data of the encounters of the term in the document. Subsequently, one would apply a merge strategy similar to conjunction with additionally filtering out documents that have no subsequent hits. Alternatively, an approach might be to add all subsequent word combinations to the reverse index (making the index significantly larger). The positional data is then not required, though. For phrase queries of three or more terms, different such index entries need to be merged and verified against the original documents in order to prevent "to be or" from matching "to be something, be or".
- **Wildcard matching** (`ca*or*ia`) can be done more efficiently using a separate tree-based dictionary structure, and additionally *permuterm* or *k-gram* dictionary indexes. Trailing wildcard queries benefit mainly from a tree structure in the dictionary. Permuterm indexes, in which an *end of word symbol* is introduced and all permutations of the word are saved, can handle all wildcard queries, but often

explode the dictionary size. K-gram limits the key size of permuterm, at the costs of more lookups for queries, joining and an additional post-processing step.

As the above shows, the exact encoding of the document list in the inverted index depends on the type of queries that are to be supported and time and space requirements.

The index can be sorted by either a unique document identifier, or some other semi-unique property such as global relevance score (this refers to some property that indicates the order it should be presented in the search results, such as a webpage's PageRank used by Google [Brin and Page 1998]). In case of any type of positional (i.e. phrase) support, the positional data has to be added in the document lists, making alignment harder due to the data's variable length (e.g. a term may appear multiple times in a document, so the postings list has no length bound). An option is to amend the document list with skip pointers; a single level skip list structure with $\sqrt{|docs|}$ distributed pointers is suggested [Manning et al. 2008].

Multiple algorithms exist to find intersections or unions of posting lists, and their applicability and performance depends on whether the lists are sorted, *aligned* (indicating that list items always take up a fixed amount of bytes) and how many lists are to be joined [Barbay et al. 2009].

To sacrifice some processing power for space complexity, posting lists can be compressed [Bell et al. 1993]. Besides using a common block compression scheme such as deflate compression, the integers that are used to represent the sorted key in the index record (document id or relevance score) can be saved using delta compression. Experiments using variable length encoding for these identifiers show significant space savings [Bell et al. 1993].

2.2 Data isolation and consistency

Arguably, the holy grail in traditional, relational database management systems (*RDMS*) is finding a *query planning* approach that allows non-blocking concurrent *transaction* execution, while guaranteeing complete *isolation* between all transactions [Gray and Reuter 1993].

Since no such miracle algorithm exists, well-known RDMS systems (e.g. MySQL [MySQL 2005] or PostgreSQL [Douglas 2005]) resolve the tradeoff between *isolation* and *performance* by letting users choose the appropriate *isolation level* over the data these systems hold (for instance through a *lock granularity* configuration parameter) [Schwartz et al. 2008]. This often involves complex lock structures and resolution semantics, without

giving any performance guarantees (or the guarantee of a successful execution, for that matter) on specific transactions.

Eventual consistency refers to those systems in which read operations after write operations are not guaranteed to yield the most recent value (i.e. they may return *stale data*), possibly even in the same transaction that did the write.

2.3 Index segmentation

A few, non-relational data stores such as Google BigTable [Chang et al. 2008], Apache Cassandra [Lakshman and Malik 2010] [Ellis 2011] and LevelDB [Dean and Ghemawat 2012] employ so-called *SSTables* (sorted string tables) to improve write performance on media that has slow seek times, such as spinning hard disks.

In principle, SSTables work by grouping updates into *tables* (or *segments*) performing in-memory updates on a single *active segment* only, flushing this segment sequentually to disk and starting a new segment at given intervals. The active segment in such systems is able to serialize any write operation (including operations such as *delete*), since non-active segments cannot be written to. Read operations are done by incrementally trying older segments, starting at the newest and reading progessively older tables until the value (or delete operation) for the key is found.

As a consequence, the most costly read operation from the store requires accessing all segments (namely when a key is chosen whose latest update was written in the oldest segment). To prevent those read operations from getting too costly, the systems can opt to *compact* multiple segments at certain triggers (e.g. "there are more than x segments" or "every 120 seconds").

Lucene, a Java library implementing full text indexing and searching, uses a similar technique to maintain its inverted list index [Hatcher and Gospodnetic 2004]; for *keys* and *values* it has *terms* and *inverted lists*, respectively, and a merge operation for similar keys consists of taking the *union* of both lists. Additionally, Lucene uses this segmentation to implement concurrency - allowing multiple readers and a single writer:

- Readers from the index will only consider the non-active (static) segments when looking up term lists (in effect being *eventually consistent*).
- A single writer can be writing to the active segment. Upon commit the segment is marked non-active and the writer starts a new segment.
- The merger follows the semantics of a reader and

- starts merging some non-active segments into a single segment;
- once merged it atomically updates the global list of segment references;
- once the last reader has stopped reading from the old, unmerged segments, they can be removed.

2.4 Database distribution

When data volumes outgrow a single system, database *distribution* provides mechanisms to split up (or, *shard*) the data among different nodes (*shards*). One way to achieve such distribution, is through *hash code sharding*, requiring a hash function h that maps the data space to a linear space.

Given such a function h, assuming its range is $[0, 1\rangle$, it is trivial to construct a mapping to an arbitrary amount of shards: $n(\delta) = \lfloor h(\delta) * N \rfloor$, where N is the amount of available nodes, δ represents some granular unit of data depending on the type of database (e.g. a table row, key/value tuple, etc.), and function n yields the shard number the piece of data should be placed.

The distribution of the hash function ought to be uniform to achieve a equally sized shards.

Generic distributed database systems that employ hash code sharding (e.g. Cassandra [Lakshman and Malik 2010], MegaStore [Baker et al. 2011], Riak [Klophaus 2010], HyperDex [Escriva et al. 2011]) often extend this basic model, for instance by any of the following approaches:

- **Map multiple shards to a single host.** By mapping multiple shards to any single host (or by using real - non-integer - shard identifiers), adding or removing hosts from the cluster can be done simply by remapping some shards to a different host. Additionally, duplicating more granular shards over different hosts in a cross-over fashion allows for balancing the load of any failed node among the others.
- **Maintain the order of a primary key in the distribution.** When using a primary key (e.g. of a table row, or the key of a key/value tuple), the distribution function could maintain the lexical order of this key. As a result, each shard will serve a range of rows (allowing prefix searches, ranged queries, etc.). Consequently, the sharding space will almost definitely be non-uniform, but mechanisms such as *consistent hashing* exist to mitigate [Karger et al. 1997].

- **Let the application specify supergrouping.** By letting the application specify supergroups of data, and using these as most significant denominator in the sharding space, data within the same group can be distributed on the same host (or on closely located hosts). MegaStore uses this, and allows relatively strong consistency guarantees within supergroups, but not between them [Baker et al. 2011].
- **Use multiple properties for the distribution.** When considering data items with multiple properties (e.g. table rows), this approach would use them all to map to a predictable spot in the distribution space. When querying for data items, the constraints on any properties can be used to narrow the set of hosts that should be queried. As a consequence, data items would quite likely have to move when any of their properties change. HyperDex employs this [Escriva et al. 2011].

2.5 Query pipelining

In term distributed fts-systems, queries may be executed using the *pipeline* paradigm: the nodes that contain the inverted lists associated with the terms in a query are configured to process (or, *accumulate*) intermediate result sets in a pipeline fashion until the final result set is obtained [Moffat et al. 2007].

In a pipeline that supports matching on a combination of terms, the intermediate result sets consist of an ordered set of document identifiers - while each nodes applies a *sorted merge join*.

Figure 2.1 demonstrates the pipeline paradigm, by illustrating how a three term query is executed.

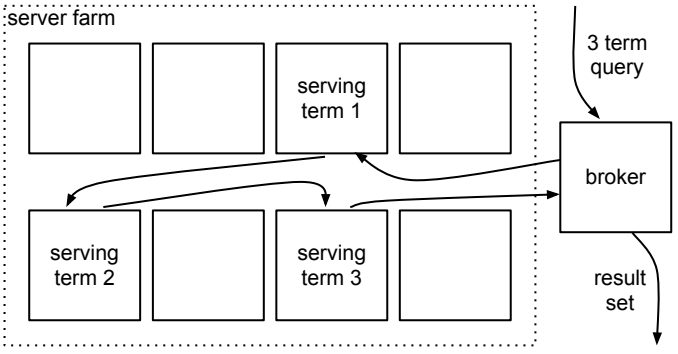

FIGURE 2.1: Example pipelined query over term distributed index.

To prevent intermediate result sets in a pipeline from overusing network resources, it is suggested to introduce an *accumulator limit*, an integer indicating the maximum number

of document identifiers in these sets, on the order of magnitude of 100.000 documents [Moffat et al. 2007].

Chapter 3

Search at Spotify

3.1 Spotify client

Figure 3.1 shows the version 0.8.4 of the Spotify desktop client, just after startup. Finding music through search is a common use pattern, and the search field is prominently shown. After executing a search query, the user is presented with an interface similar to that shown in figure 3.2: artists, albums, public playlists and individual track matches are shown.

3.2 Infrastructure

The Spotify search service is a loosely coupled service implemented around Lucene.

The majority of its requests originate from user-entered queries in a Spotify client. Separate Lucene indexes are maintained for artists, albums, tracks and public playlists and these are accessed in parallel for query requests. The total number of searchable entities (documents in Lucene terminology) in the indexes is around 30m (in part because of the differences in geographic availability there are significantly more than 15m track entities). Indexes are kept in-memory, and take up around 7 gigabytes. Although most metadata fields (e.g. track artist, track genre, playlist author) are *indexed* (the document identifiers are added to the inverted lists of the terms occuring in those fields), the records themselves are not *stored*: the original document cannot be retrieved from the index and, consequently, the search service only yields document ids for queries; another backend service is used to map these to the document's metadata.

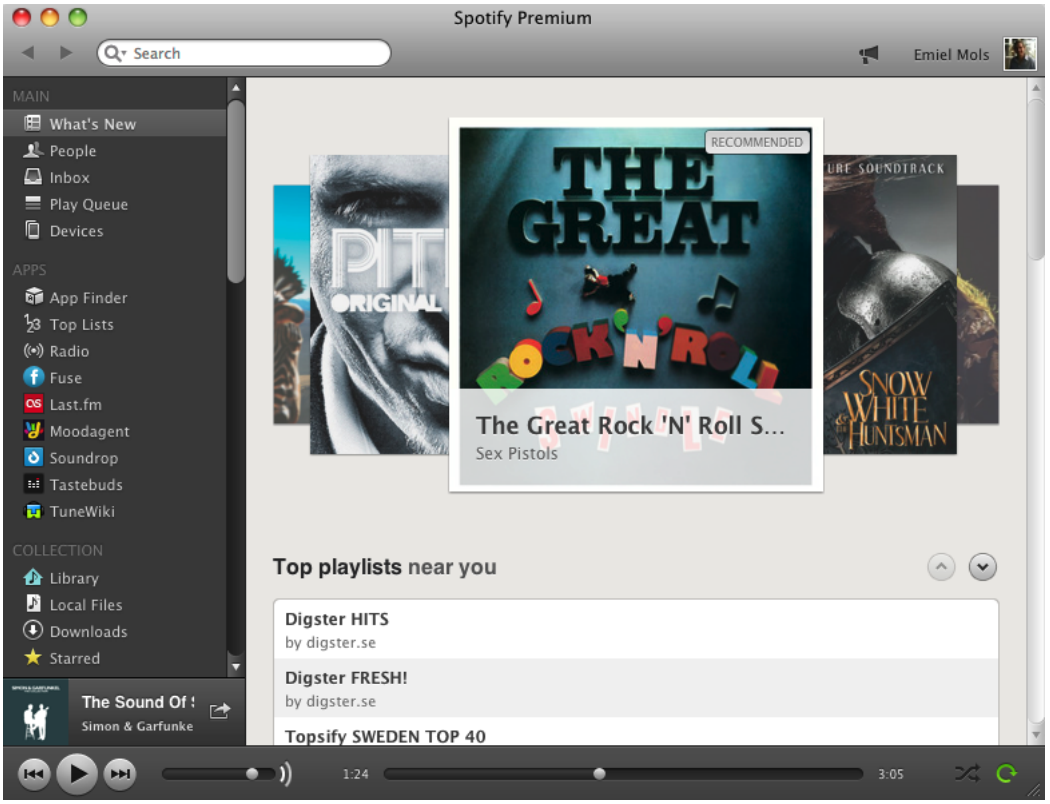

FIGURE 3.1: Interface after startup

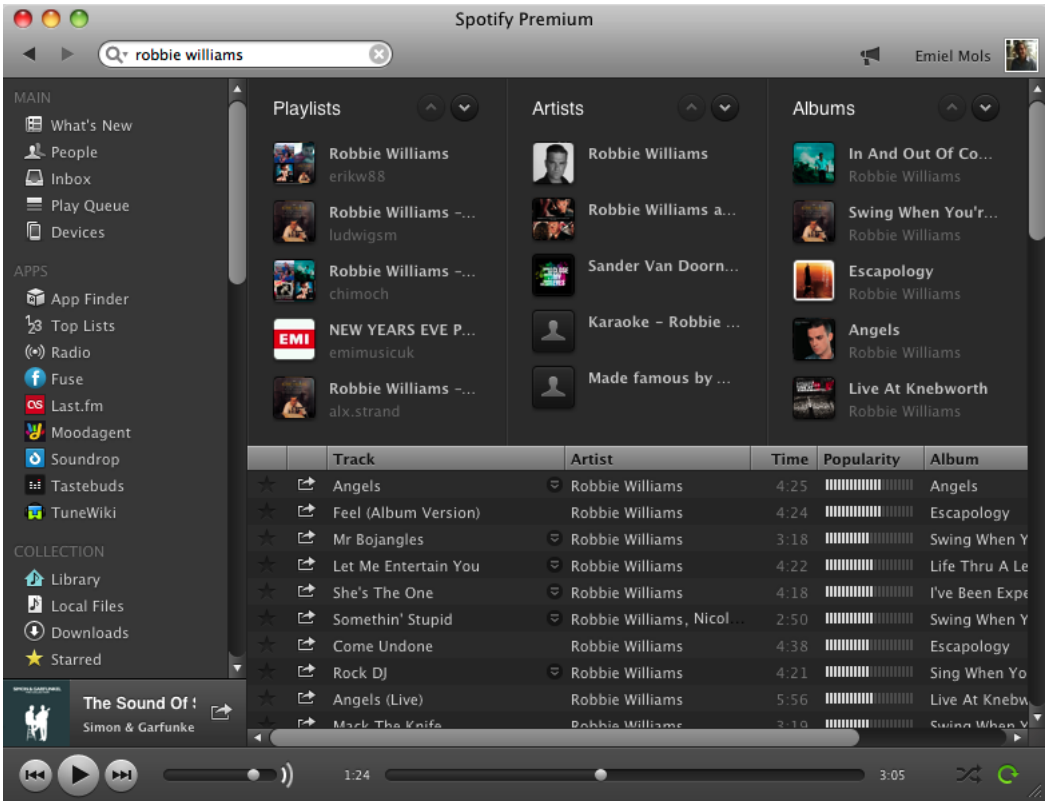

FIGURE 3.2: Interface after doing a search query

Due to different availability restrictions of content based on geographic regions and subscription status of the user, an additional filter is applied before serving the results, filtering out content unavailable to the requester.

A global popularity score is associated with each item in the different indexes, used as a significant factor in ordering the search results. It is combined with Lucene's built-in approach to reorder based on similarity to the original query.

Indexes are built and distributed in a daily batch; any changes between those batches are distributed and applied as live changesets on the index. The daily batches are used for robustness (eventual recovery from possibly failure to apply some changesets), a baseline for the changesets and a logical point where any index structures might be compacted. The amount of changes (added or removed documents) per day is in the order of tens of thousands.

Every 7 days, new popularity scores are calculated for all the entities in the collection, in effect changing all the documents in the index.

The total amount of queries served per month is in the order of magnitude of hundreds of millions, and scaling is handled through service replication and round robin distribution of requests.

3.3 Usage analyses

Performance of any database system is largely dependent on user behavior, and to allow sensible reasoning about the system's performance, we present a number of different analyses, performed on the complete search query log of November 2011.

- **Query frequency distribution**. Figure 3.3 shows a cumulative distribution (*cdf*) plot of the *the unique queries* versus the *total query volume* in the sample set, indicating the repetitiveness of certain queries (for example, 75% of the total query volume consists of approximately 0.85% of the unique queries). The *long tail* shape of this graph is as expected. The repetitiveness is presumably higher than that of web search engine systems: in 1999, the 25 most common queries made up 1.5% of the query volume of the AltaVista search engine [Silverstein et al. 1999]; for Spotify, the top single query constitutes almost 2% of the volume.
- **Query cachability distribution**. We analyzed the cachability of the queries by pulling in $105x$ query records from the query log stream: the first $100x$ were used to construct a mapping of query to query age (query 0 has age 1.00, query $100x$ has

age 0.00), where x is in the order of magnitude 2^{27}. For each of the remaining $5x$ entries, the map was consulted to determine how long ago the query was performed before (if not found, the age becomes 1.00). Its distribution is shown in figure 3.4.

- **Inverted list size distribution**. By counting the occurence of different terms in the search *corpus*, we estimate the size distribution of the different inverted lists. The distribution is shown in figure 3.5; as shown, the most common terms (the first three are, in order, `the`, `of` and `in`) take up a signifant amount of the inverted lists sizes (when they would not be *stop words*).

- **Popularity distribution**. The primary sorting variable used for sorting results is a document's *popularity*. Figure 3.6 shows the distribution of this constant over the *track* collection.

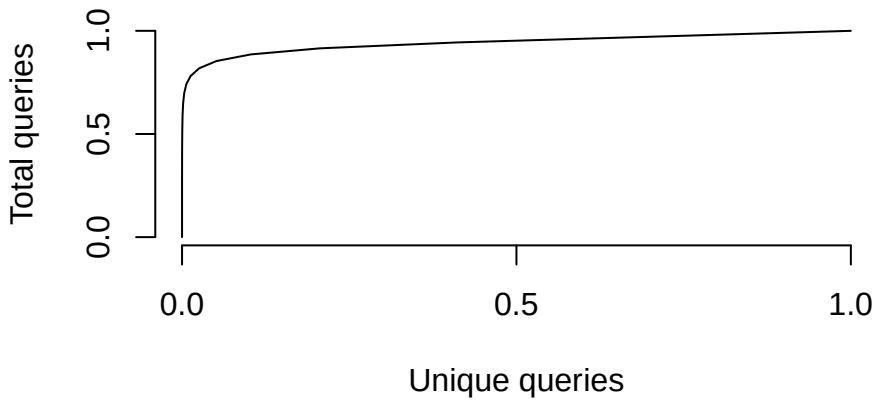

FIGURE 3.3: Query frequency distribution

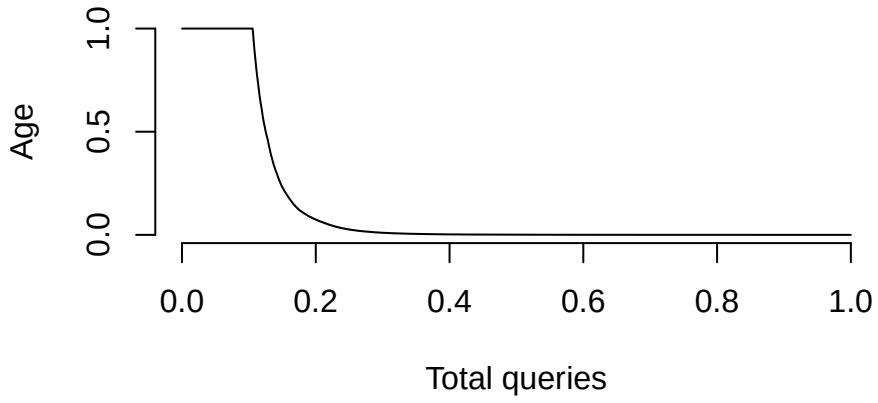

FIGURE 3.4: Query cachability distribution

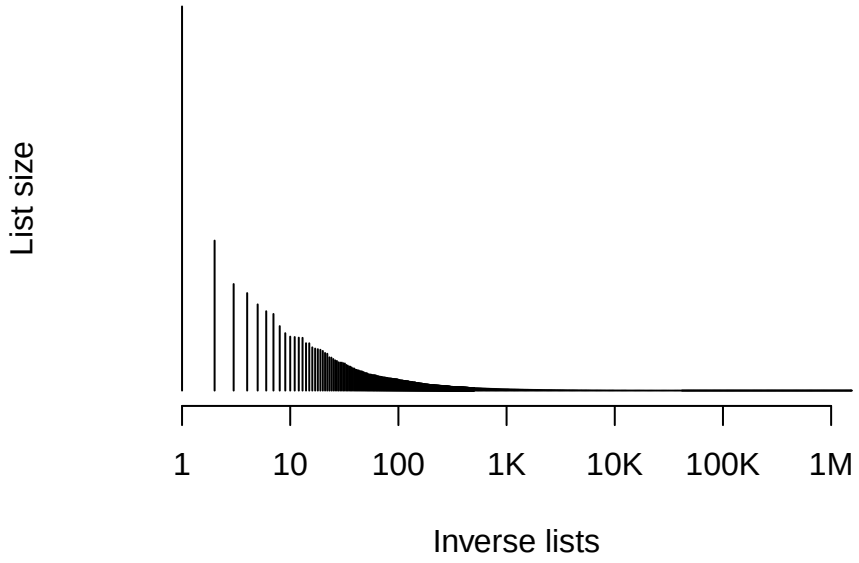

FIGURE 3.5: Inverted list size distribution

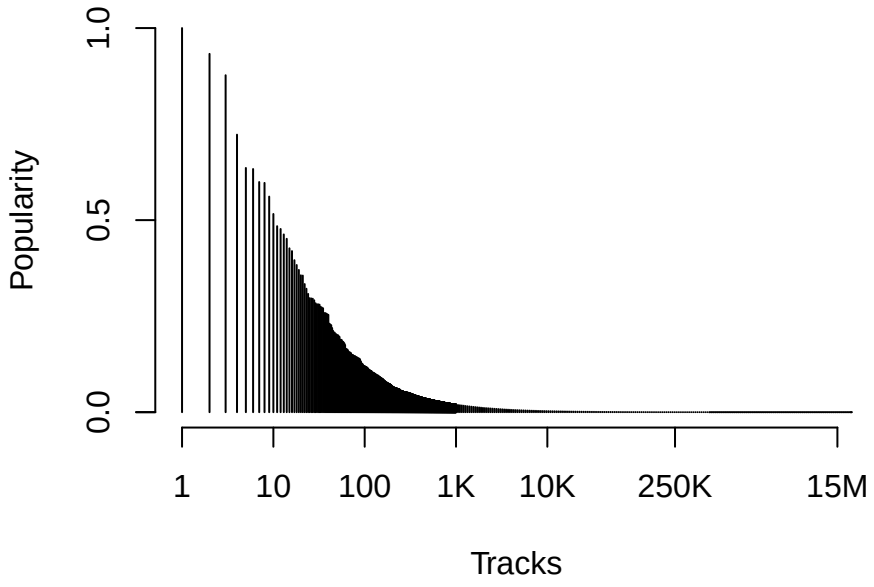

FIGURE 3.6: Popularity distribution

Chapter 4

swarm: a term distributed full text search service

This chapter presents the term distributed fts-system we implemented for this research, dubbed `swarm`. Additionally, we highlight three unconventional techniques used:

- **Enhanced pipeline model.** We describe a pipeline model that allows arbitrary boolean query expressions, herewith improving on the model described by Moffat e.a. [Moffat et al. 2007]; see section 4.4.
- **Index segmentation through forked read process.** We separate each shard into a *read* and *write* process and use Linux (or Darwin) *copy-on-write semantics* to minimize write contention on a shard without adding to program complexity; see section 4.5.
- **Inverted list subdistribution.** To mitigate the problem of unbalanced shards, we distribute large inverted lists among different shards; see section 4.7.

4.1 Scope

Because of the scale of the project, the prototype we present will have a series of functional limitations.

- **High availability:** Replication and dynamicly moving around individual shards based on resource usage is not implemented.
- **Caching:** Updating local caches at the different shards is not implemented.

- **Complex query planning:** The query to *query plan* conversion, as we will elaborate in section 4.4.1, is rudimentary.

4.2 Architecture overview

The two main design criteria for the prototype of our term distributed full text index, based on the premise as given in section 1.2 and the current system design (section 3.2), are as follows:

- **Optimize for read.** The architecture should be highly optimized for read performance, where write performance comes secondary. This is similar to one of the design philosophies of MegaStore: *reads dominate writes in our target applications, so it pays to move work from read time to write time* [Baker et al. 2011].
- **In-memory storage.** The cumulative amount of available main memory over all the shards is in the order of magnitude of hundreds of gigabytes on modern hardware, even with only a few nodes in the cluster. We therefor opt to keep the entire index in-memory. Besides the latency and throughput gains, using main memory only should lower the latency variance and improve relative random access performance (requiring less data locality of the algorithms we use). This is similar to Google's web index since 2004 [Dean 2010].

The cluster architecture consists of three distinct, orthogonal services.

- **routerd** *(one per host)* Keeps transport links to all other routers, and routes messages both to shards in the cluster both on the local and remote hosts.
- **queryd** *(one per host)* Parse query and construct a query pipeline to execute on the cluster. Waits for all responses from the cluster, and forwards the result on to the requesting client. Possibly forwards results to appropriate shards if they need to cache it.
- **shardd** *(multiple)* Does the legwork by executing the pipeline, or a part thereof. If the complete remainder of the pipeline is executed at this node, the result is sent back to the appropriate *queryd* instance. If only a part is executed, the intermediate result and remainder is forwarded to a shard responsible for the remainder.

Figure 4.1 shows the differences in a sample setup (4 hosts, 64 shards).

4.3 Data transport

As a data transport between the `swarm` services, we use ZeroMQ, a message queueing library. ZeroMQ exposes an API not unlike POSIX sockets, but implements some higher-level message exchange paradigms (e.g. `request/reply`, `push/pull`, or `publish/subscribe`) [iMatrix 2012].

All services communicate through the local `routerd` process over Unix named pipes. `routerd` maintains outgoing `push` sockets to all other routers over TCP. Doing local communication through `routerd` allows atomic *hand overs* when new shardd processes are forked (due to the concurrency model we use, new processes are *forked* at regular intervals - see section 4.5).

ZeroMQ allows sending data from arbitrary buffers, without copying the data first (*zero copy*). Because the list chunks in memory are formatted similarly to how they are sent between nodes (see section 4.6), we use this to prevent unnecessary data copying or transformation. For read (*merge*) operations, the list data is only copied when traversing the *user-kernel space barrier*: once when sending the data (data is copied into the kernel's TCP *send* buffer) and once when receiving (data is copied from the TCP *recv* buffer).

An overview of an instance of the system noting the involved ZeroMQ socket types and data flow is shown in figure 4.1.

4.4 Query pipelining

To merge the document lists at different shards, a pipeline architecture, as suggested by Moffat e.a (see section 2.5) is a good fit in many ways:

- **Simple and stateless.** It allows asynchonous operation in a *merge and forget* manner: shards handle the first item in a merge queue, pass it on to the next shard and are done. Merge operations could even be split up to different hosts serving the same shard, similar to the packet switching paradigm in networking. Furthermore, the processing of a single query is inherently distributed and the possibility of dividing up the work in arbitrary small units should, theoretically, allow for more effective work queue scheduling and subsequently preventing varying workloads or load congestion.
- **Latency optimized.** By limiting network roundtrips as one would incur in a request-response model for the list-serving shards to a central broker.

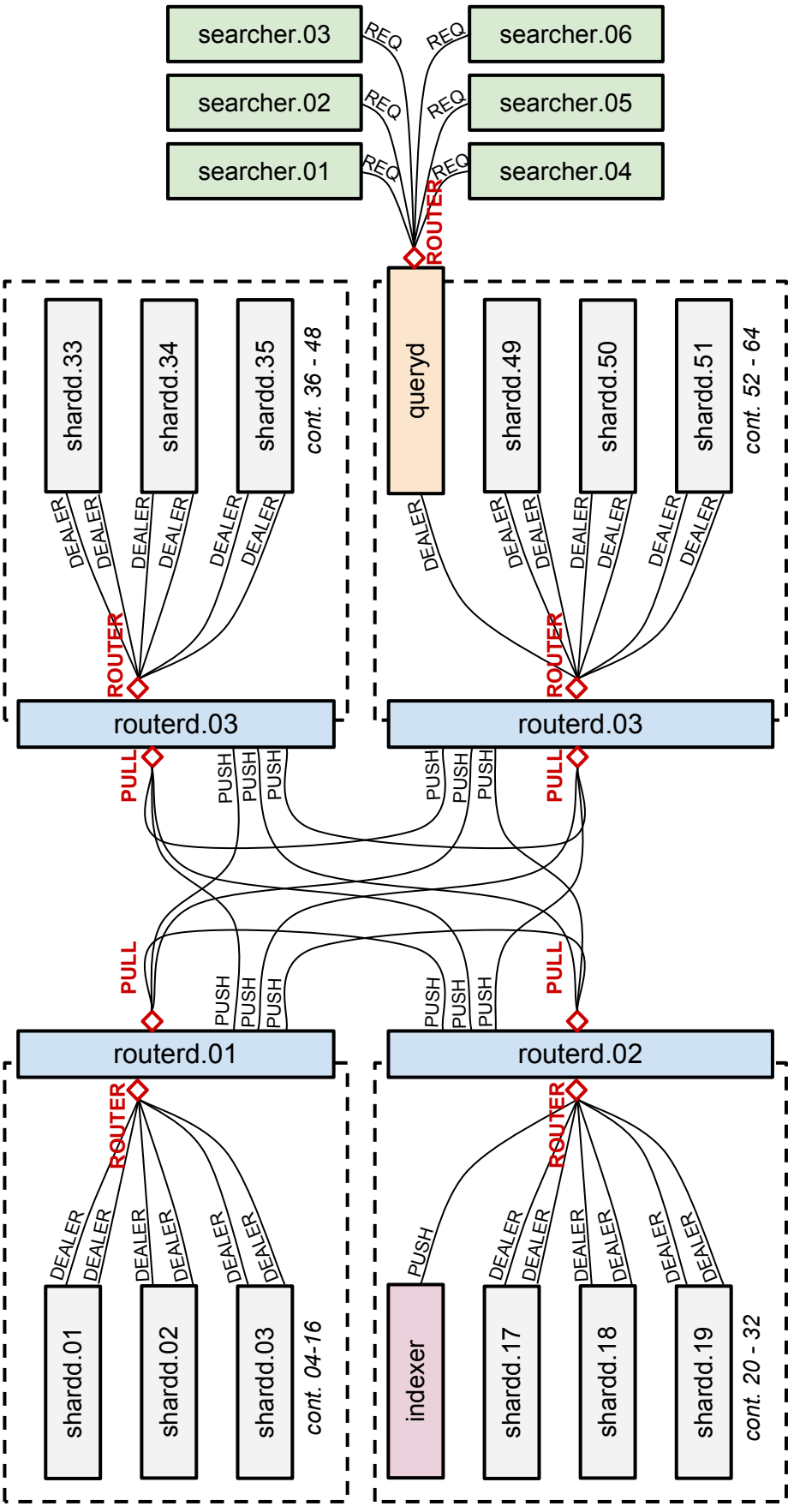

FIGURE 4.1: ZeroMQ architecture of swarm instance

- **(Partial) cachability.** Workers in the pipeline might choose to cache a combination of lists and use this cache to shorten subsequent query pipelines.

The basic query pipeline (dubbed a *query bundle* by [Moffat et al. 2007]) as described in section 2.5 can be regarded as a simple list, where each list node (specifying a *pipeline node*) contains the *term* (which implies the *shard*). Execution of the pipeline starts at the shard specified by the *tail* element of this list, and ends at the shard specified by the *head* element.

We extend this model by associating the following data fields with each pipeline node:

- **term**, the term associated with this operation, as present in the simplified model;
- **level increment**, integer indicating depth increase of the associated query tree;
- **instruction list**, where each instruction contains an
  - **instruction**, indicating the type of *merge operation* (see below for some example types);
  - **level decrement**, indicating the depth decrease of the associated query tree for this operation;
- **result cutoff**, indicating after what amount of results the shard can stop processing.

The proposed extension consists of two parts:

- Specification of multiple and different instructions for each pipeline node. The instruction field specifies the *sorted merge join* filter to use (e.g. *intersect*, *union*, *intersect for specific field*, *intersect if words form a phrase*).
- The *level delta* fields, that specify a *level* value for every shard (and every instruction at every shard). Over all the shards, the sum of all the *level increment* should match the sum of all the elements of all the elements of *level decrement* values. Using these integer values, we support querying for arbitrary boolean expression trees.

A query pipeline list is formed by a depth-first traversal of the applicable boolean query tree. As an example, we consider the query `"A B" or (C and D)`; its query tree is shown in figure 4.2. The associated pipeline specification, and a block diagram of the execution by `swarm` is shown in figure 4.3.

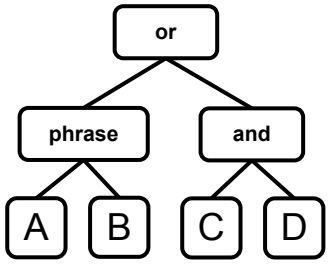

FIGURE 4.2: Example query tree

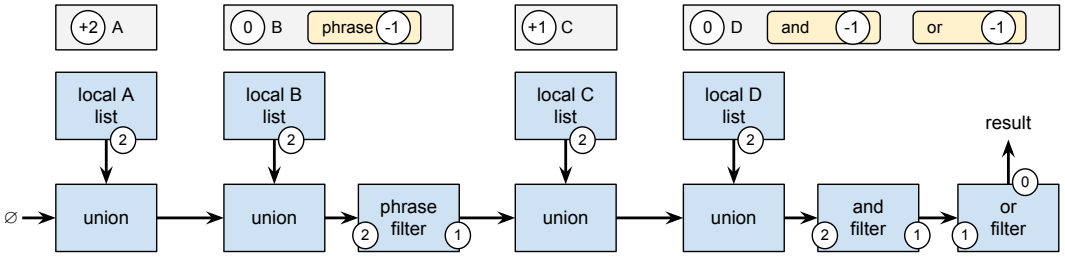

FIGURE 4.3: Example pipeline and execution diagram

The gray blocks in figure 4.3 indicate different pipeline list nodes, the integers in the circles inside indicate the *level delta* values.

In the lower part of the diagram, the arrows represent a stream of (docid,level,hits)-triplets between the blue operation blocks, where an order based on (docid,level) is always maintained. The operation blocks are marked with numbers that signify the input and output *levels*: documents that do not have a similar *input level* are not processed (and simply passed along); matching documents are processed and ultimately marked with the *output level*.

The *and filter* matches if two subsequent document identifiers are identical. The *phrase filter* can be regarded as a specialization hereof: it only matches if two subsequent document identifiers are identical and the *hits* value of both instances indicate there is a document field where the two terms occur at a specified offset of each other.

4.4.1 Query planner

The `queryd` processes take care of *query planning*: converting user queries to a query pipeline specification. After parsing a query tree, any operations in the query pipeline can be *reordered* based on an efficiency prediction of the query plan. The prediction heuristic uses the globally shared *term-to-distribution mapping* (see section 4.7) and (to a lesser significance) word length of the terms associated with the operations. Operations with terms that have a higher cardinality, or have fewer characters (implying a lower

cardinality, on the premise that longer words generally are less common in natural language) are pushed to the back of the pipeline list.

4.5 Concurrency

As explained in section 1.1.2, a property of *term distribution* is that the sharding space can be split up indefinitely without negatively influencing query performance. As a consequence, we can keep shards small, and allow

- a single-threaded *shardd* process, because the many other shards on a single host will make it likely load is distributed on multi-core systems;
- more granularity when load balancing through replication or moving shards between hosts, because it is cheaper to transfer a shard's dataset.

Each `shardd` separates reading and writing into two separate processes: when launching, a writer process is spawned. The writer process `fork()`s a new reader subprocess at fixed *rolling intervals*; when a new reader is spawned, it signals the previous reader to stop operation.

The (Linux and Darwin) `fork` call has *copy on write* semantics: the operating system will mark the memory pages in the writer's process space as read-only, only physically copying a page when it changes, and subsequently mapping the virtual memory transparently. This makes launching the reader process cheap (as opposed to copying the all the physically mapped memory in the process space directly!), and only requires so much memory overhead to accomodate changes. This approach bears resemblence to how key-value store Redis persists snapshots of its data on disk: it forks a subprocess that simply does a linear scan over the subprocess' view of the data and dumps this to disk, while, the main process can continue updating data structures in memory uninterrupted [Sanfilippo and Noordhuis .

This approach introduces two major benefits:

- **Write operations do not significantly worsen query latency.** Because of the separated processes, read and write operations can occur simultaneously. In case all write operations are small atomic operations, the advantages do not show directly and one could use a single process to handle both reads and writes in a single queue (as Redis does). However, we also use this mechanism to execute a more complex *optimization* step at the end of each cycle.

- **No application level locking.** `shardd` uses no locks at all, greatly increasing application simplicity. Complex locking procedures are executed by the system's kernel or the hardware itself (e.g. most machines use hardware-assisted TLBs that will atomically remap virtual memory).

As a trade-off, changes to the index are not visible immediately, but - worst case - after the rolling interval time. Even combined with the behavior of merge operations between different shards, this results in an *eventually consistent* view of the index.

When comparing our approach to Lucene's (as detailed in section 2.3), we see that both offer *eventual consistency*. However, we believe that our approach offers more flexibility in data optimization strategies: the writer process could change (i.e. optimize) whatever piece of data it sees fit, at any time, and data is only duplicated at the granularity of page sizes. If Lucene would want to change anything in the non-active segments, it would have to make a complete segment copy. Furthermore, `shard`'s implementation in this regard is much simpler, because the logic to atomically keep track of the segments is not required.

Of course, our approach is limited to in-memory operation, while Lucene can use any storage medium.

4.6 Storage

An *unrolled doubly linked list* data structure is used to save parts of the inverted lists in their *wire format*. Unrolling entails saving a set of sequential items with a single set of list pointers. In our implementation we align these chunks with the system's page size. It significantly decreases the pointer overhead per list item, increases data locality when iterating over the list. Combined with saving list chunks in the same binary format that is used to transfer them over the network, this allows complete list chunks to be pushed on the network stack when executing queries.

A consequence of aligning the list structures on page boundaries is that a large amount of short lists (see section 3.3) would have an excessive memory footprint: a page is commonly 4kb large and very small lists rarely larger than 100 bytes. To mitigate, we merge those smaller lists (called *tail merging*) into a single page at the *collapse* step.

All the data contained in the `shardd` processes is contained in a single contingent block of virtual memory, allocated using the `mmap()` system call. The linked list pointers use relative offsets from the memory mapping's base address. This allows the entire data

blob to move to a different virtual address region (e.g. when moving to a different host), and to be persisted to and reloaded from disk without any processing.

For the *vocabulary* structure, C++ stdlib's `std::map` implementation is used, which uses a red-black tree internally.

The effect of tail merging is shown in figure 4.4, which shows the cumulative allocated block size over 2 `shardd` process instances - one where tail merging is enabled, one where it is disabled. Over the x-axis, 1M Spotify track documents are added to the index. The rolling interval is set to 20 seconds, and the sharding factor (i.e. shardd processes used) is 4.

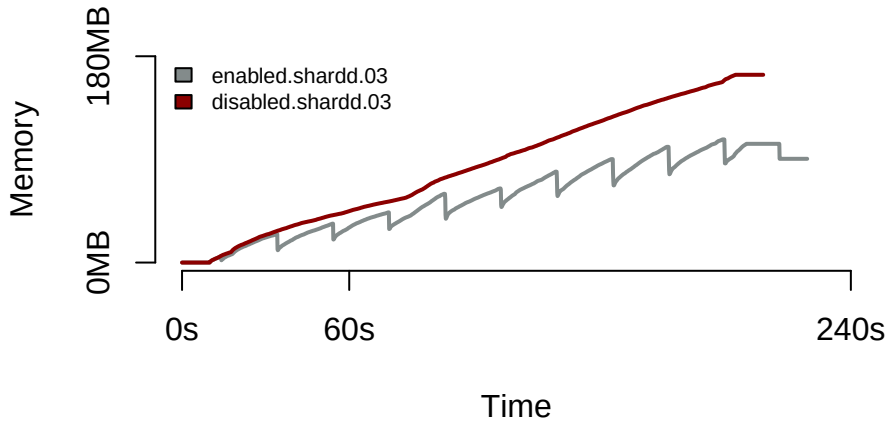

FIGURE 4.4: Effect of cycled tail merging on memory consumption

4.7 List subdistribution

The inverted lists are distributed through the most elemental form of hashing the list's key (term), as described in section 2.4. Most advanced features as seen in other database systems and discussed in section 2.4, seemed either unnecessary or not applicable: there is no natural *supergrouping* in a set of inverted lists; ordered traversal through different inverted lists is unnecessary and there are no secondary properties that could be used to form a *hyperspace*. We do map multiple shards to a single physical host, but have no

As we showed in section 3.3, the distribution of inverted list sizes is unbalanced. To mitigate the problem of unbalanced load when sharding solely on the list key, `swarm` includes a mechanism to *subdistribute* larger inverted lists.

It keeps track of a *term-distribution* mapping on all nodes and query services. The distribution indicates how many *sublists* a list is split into. Additionally, a sublist identifier is added to the inverted list key (and, as such, used by the sharding function). Whenever a list is considered too large for the current distribution factor of that list, all nodes that serve a part of the list split their lists and redistribute a part of it (quite likely, but not necessarily, to a different shard). This term-distribution mapping is shared among all nodes (we sacrificed *shared nothing*). Since only terms that have splitted lists need to be saved in this map, the mapping will use a relative small (although not constant) amount of memory. Its size complexity is $c * sqrtn$, where n is the corpus size (we assume natural language and $sqrt(n)$ as the number of inverse lists, see 2.1. It is hard to formally find bounds for c: its value is dependent on the *split threshold* that is chosen. In our practical setups, its value has been approximately 0.5 (e.g. a *corpus size* of 20M documents with a *split threshold* of 30000 results in a size of the mapping of around 250).

In the execution phase of a query pipeline, the distribution mapping is checked to see if the subsequent execution should be split to different shards. To illustrate this process, figure 4.5 shows an example pipeline execution of the query `the black keys`, where the term `the` has a granularity integer of 4. The thickness of the lines between the nodes indicate possible (sub)result set sizes.

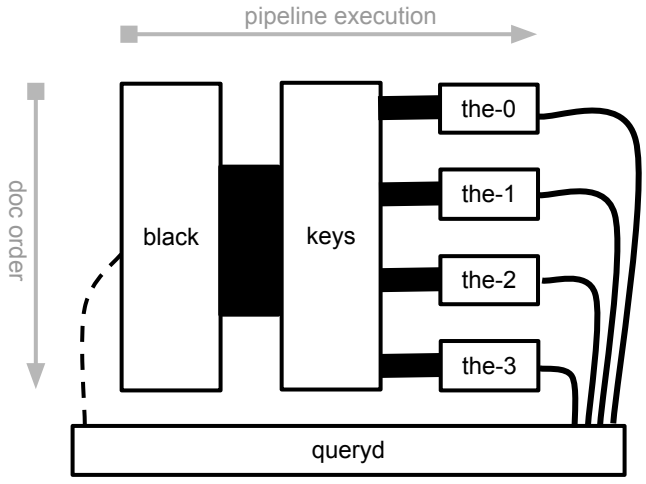

FIGURE 4.5: Example query execution

Lists are split up through linearly dividing the absolutely ordered document space: all documents have a unique ordering identifier, that we assume to be linearly distributed for all documents in the system. For an 8-way distributed term, for instance, sublist i serves the document range $[(i-1)/8, i/8\rangle$, assuming a total document range of $[0, 1\rangle$. As a consequence, we require all document ordering values (which, in Spotify's case,

might be the global popularity value of a track) to be uniformly divided over this space (currently not the case; see figure 3.6).

The shard that serves the first part of the list (range $[0, x\rangle$) is authorative on the granularity value (in other words, it determines when parts should split). When a new value is broadcasted, shards serving parts of the lists *repel* smaller parts to other shards (and can remove the parts locally). Because of this single authority and the fact that we assume the distribution integer only increases (e.g. parts are only splitted, never merged), no *distributed locking* is necessary.

We suspect that *stop word filtering* (see section 2.1) would also have helped mitigate some list distribution issues. However, our approach to evaluating phrase queries that contain such stop words, such as `"in the middle"`, requires that all the inverted lists are available; we do no additional filtering or *query reevaluation* when we have a result set. Additionally, our approach is *content agnostic* (other systems often use language based stop word lists).

The plots in figures 4.6 and 4.7 show the effect of list subdistribution on the memory sizes of different shards, in a run of indexing 1M Spotify track documents over 8 shardd processes on a single machine with the rolling interval set to 20 seconds; the sublist threshold for the run depicted in figure 4.7 is set at 5000 documents. The additional run time needed when subdistribution is enabled, is significant: it takes almost 50% longer. However, the memory distribution between shards is a lot better with the subdistribution feature enabled.

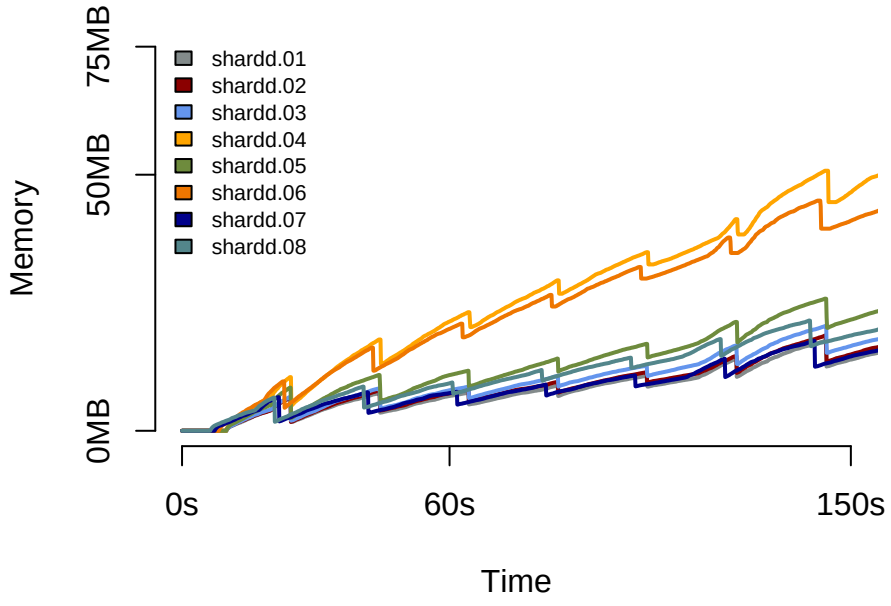

FIGURE 4.6: Effect of list subdistribution on shard sizes: disabled

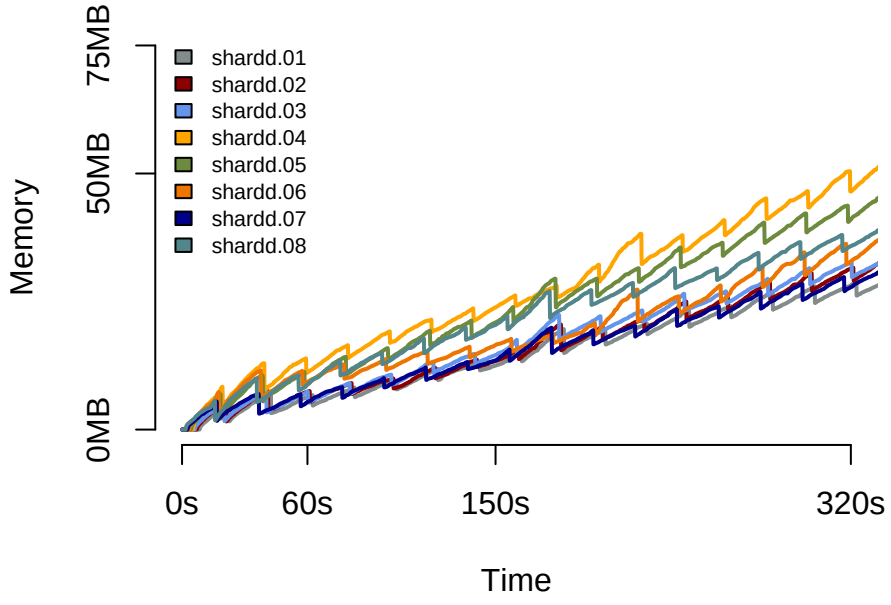

FIGURE 4.7: Effect of list subdistribution on shard sizes: enabled

Chapter 5

Performance analyses

This chapter presents a series of performance experiments done on `swarm`. For each experiment we present the rationale, detail the specific setup and present and interpret the results.

Because our prototype has become a relatively complex system, there are virtually unlimited performance experiments that one could perform. Due to the scope and associated time contraints of any research, we had to make some choices here. Generally, we chose experiments that evaluate the read performance of the system as a whole. In section 6.4, we propose some additional experiments that could be performed to quantitatively evaluate individual features.

5.1 Global setup

All benchmarks in this chapter were performed on 8 identical hosts in the Spotify backend, each with 2 quadcore Intel Xeon processors and 16GB of physical memory, connected over a gigabit network. We configured 8 `shardd` processes per instance, amounting to 64 shards in total. Each host runs an additional `routerd` and `queryd` instance. The number of 8 shards per host was chosen to map on the physical processing units available.

Because the subdistribution feature of `swarm` requires a linear distribution of the *global order* property over the corpus, we did not use the track's *popularity* property (which is non-linearly distributed; see figure 3.6), but assigned a random value over the space. Although this generates unusable results for a practical search application, it should not make much of a difference for the test results: in a practical setup one could apply some normalization function to the *popularity* property.

5.2 Load distribution

The goal of these tests is to ascertain to what degree the load of the system is distributed among the hosts we use in our cluster: we aim for a uniform load distribution, as this allows maximization of available hardware resources and implies favourable scaling characteristics. To accomplish a uniform load distribution of hosts, we actually strive for load uniformity of the individual shard processes. Our measurements are thus performed at shard granularity, where possible.

Used memory blocks. Firstly, we gauged our implementation by monitoring the amount of actual page-sized blocks used for the list chunks. We believe tracking these *used blocks* is slightly more precise than the more traditional metric of *wired process memory* (reported by tools such as `ps`), since it is independent of the system's memory allocation semantics (e.g. in practice the system could allocate larger, contingent chunks). More importantly, though, optimizing our system for specific kernel behaviour is outside the scope of this research, so measuring closer to the implementation makes more sense.

Figure 5.1 shows the distribution of this *used block* metric over 64 shards with 20 million track documents in the index. As shown, the distribution is fairly uniform. We used a *sublist split threshold* of $30,000$.

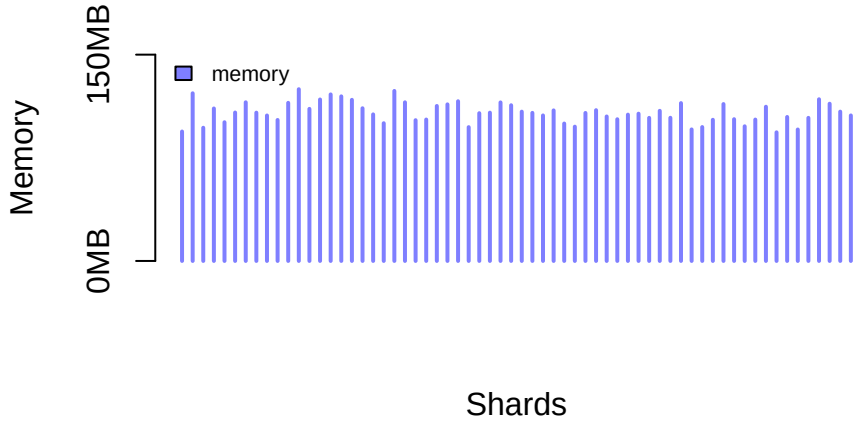

FIGURE 5.1: Memory distribution; 20M corpus

Secondly, we want to find out the distribution of load when running queries on the cluster. A traditional metric we could use here would be *CPU utilization* by sampling the kernel's process wait queue (through tools such as `uptime`). The resulting metric indicates how much of the processor's time is spent non-idle. Additionally, most kernels

keep track of the *CPU time* for each process, indicating how much cumulative CPU time the process has had scheduled to it. Both figures could be used to determine the CPU load on system or process level granularity. However, similar to the argument above (favoring measuring the used blocks over the actual wired memory), we feel there are more accurate metrics here - closer to the scope of this research, and less prone to influences from other layers (e.g. the operating system's SMT scheduling behavior).

Operation distribution. We measure the amount of *merge operations* each shard has to perform. Merge operations can vary a bit in size (e.g. can contain both a small number or a couple of thousand of documents to merge), so we additionally count the number of documents each shard outputs. Since the actual implementation has some overhead per merge operation, and additionally uses resources for each document, a uniform distribution of both figures would imply a uniform distribution of the resulting load. Figures 5.2 through 5.4 show the merge operation distribution for different corpus sizes. As the graphs show, the distribution becomes more uniform as the corpus size increases. Again, a *sublist split treshold* of $30,000$ was used.

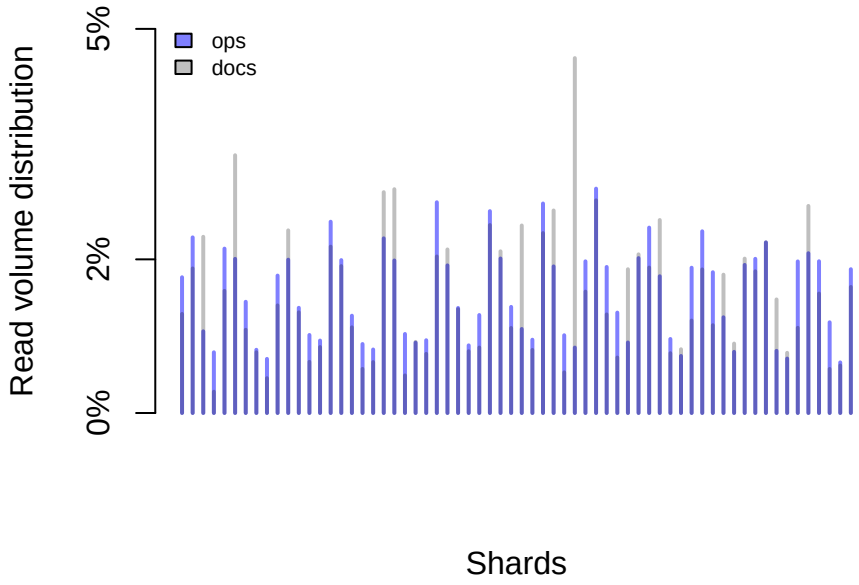

FIGURE 5.2: Read distribution; 2M corpus

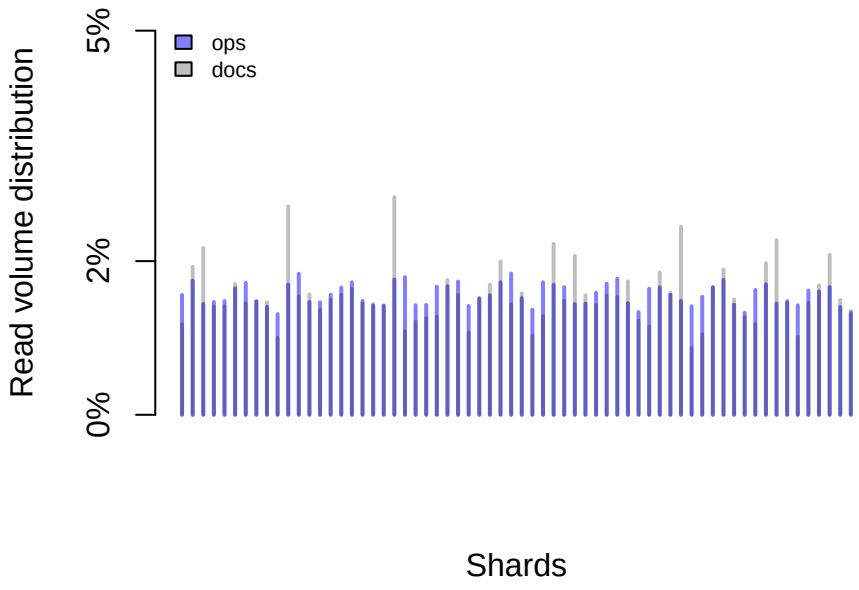

FIGURE 5.3: Read distribution; 10M corpus

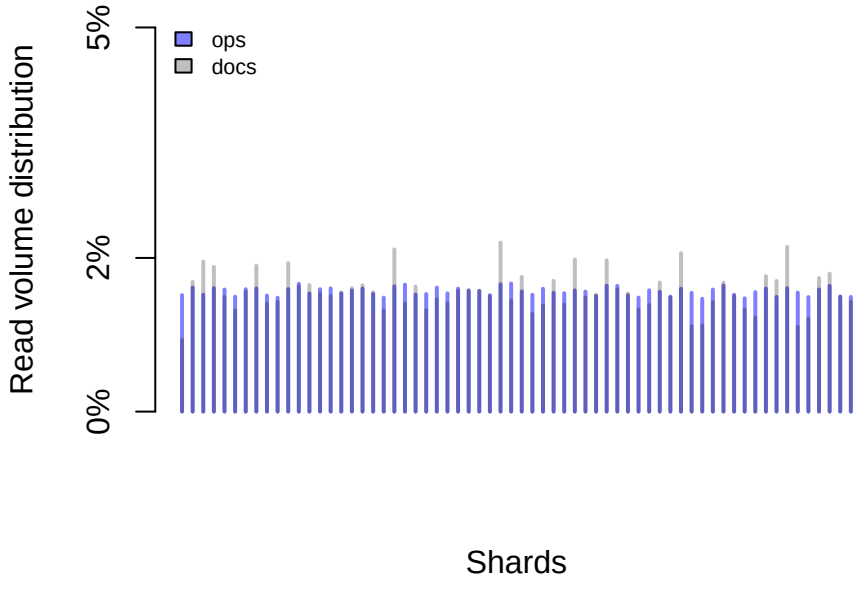

FIGURE 5.4: Read distribution; 20M corpus

Waiting queues. I/O-bound database systems that employ slower, physical storage systems often do performance benchmarking with a metric that involves the I/O's *cumulative wait time* or *outstanding queue size*. Although we do not use hard-disk I/O queues, ZeroMQ (used by `swarm`'s transport layer), uses message queues internally. Since ZeroMQ does not expose such statistics itself, we wrote a tool that can access the library's internal state [1]: we are able to sample the *backlog size*, measured in ZeroMQ messages, of a socket, indicating the amount of unhandled messages in the ZeroMQ receive buffers. Figures 5.5 through 5.7 show the relative sizes of these queues over all the shards, while replaying the 1M query log, measured by sampling the shard's communication socket's backlog size every 10ms.

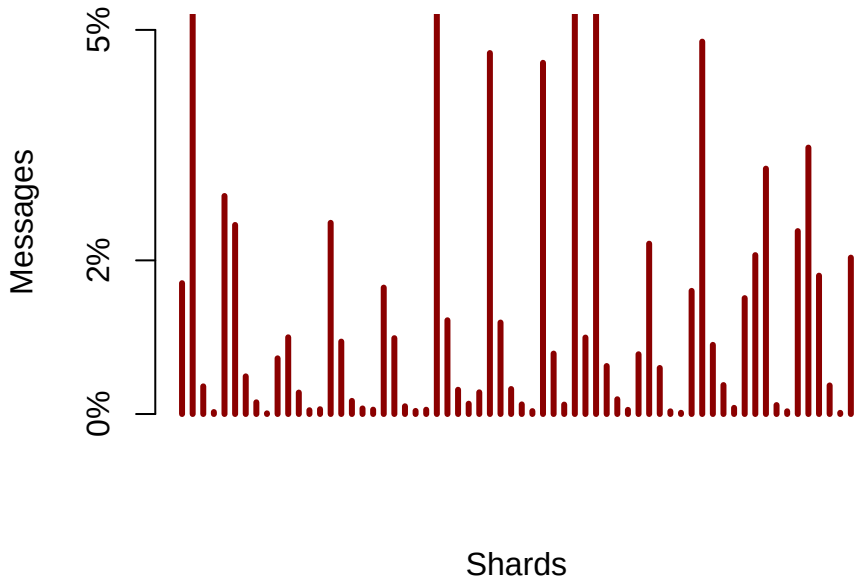

FIGURE 5.5: Message backlog distribution; 2M corpus

[1] Released under an MIT license; available at https://github.com/EmielM/libzmqperf.

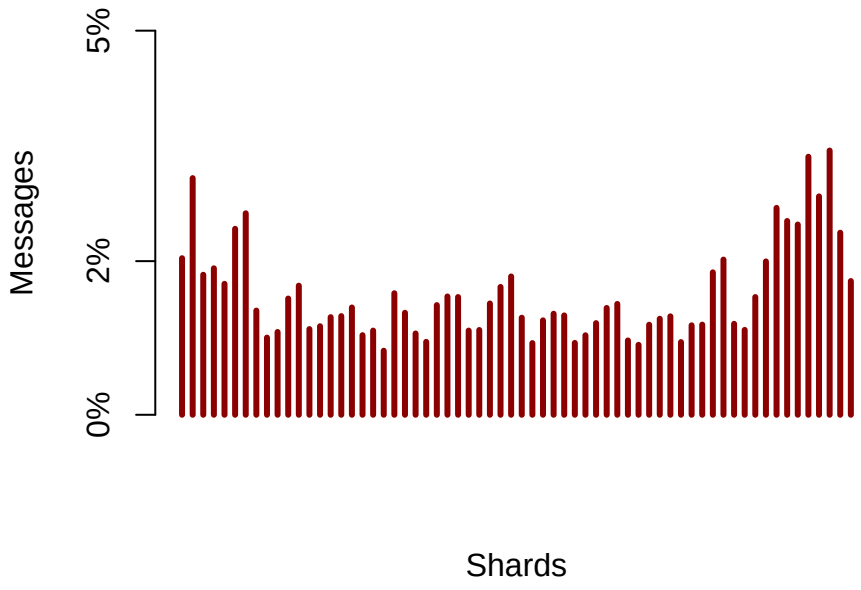

FIGURE 5.6: Message backlog distribution; 10M corpus

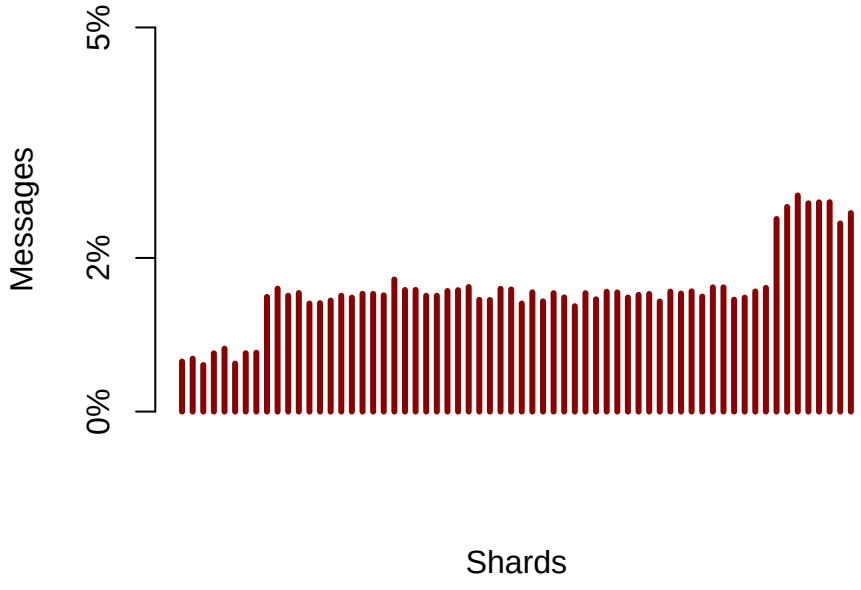

FIGURE 5.7: Message backlog distribution; 20M corpus

Both the uniform memory usage and the uniform operation distribution is exactly as hypothesized during the design of the system, and caused by the *list subdistribution* feature: as the number of split lists increases, memory usage and load distribution become more uniform. The *sublist split threshold* should be adjusted to the *corpus size* and *shard count* to achieve reasonable load distribution (e.g. in these benchmarks a value of $30,000$ is too high for the corpus of 2M documents and causes an unbalanced load).

The backlog measurements support the observations and conclusions for the operation distribution experiments (backlog size gets more uniform as corpus size increases), except for perhaps the significant difference in backlog size for shards 1-8 and 49-64 in the 20M experiment. We cannot present a viable explanation for this inconsistency, but it might be grounded in a different (hardware or software) configuration of two of the machines (a single machine hosts shard 1 through 8, a single machine hosts shards 49 through 64). Additional experimentation is necessary to determine the exact cause.

5.3 Latency and throughput

The goal of these tests is to determine the latency and throughput of the system, and which variables influence these metrics. We note that there is no *caching* whatsoever implemented in our prototype implementation, which if implemented would yield significant latency and throughput improvements: a cached result would make the pipeline only 1 host long (mainly improving latency), and prevent possibly large intermediate result sets from being transferred over the network (mainly improving throughput). Figure 3.4 suggests that more than 70% of the query volume could be served from a relatively small cache.

As first experiment, we replayed 1M user queries on the cluster, which was populated with 2M track documents, using different values for the *maximum outstanding queries* (*moq*) variable. This variable is controlled by the script that replays the query log; a moq of 256 would indicate new queries are only executed when there are less than 256 queries for which a response has not been received yet. We measured the response time for the queries, and plotted the corresponding distribution graph in figure 5.8. For the outstanding query limit we tried values 64, 128, 256 and 1024. The run times of the individual experiments is shown in table 5.1, with the corresponding query throughput.

These figures are as can be expected: a higher moq value yields better throughput, sacrificing a better (lower) latency.

Additionally, we experimented with the influence of the corpus size on latency and throughput. Figure 5.9 shows distribution graphs for corpusses of sizes 0, 2M, 10M

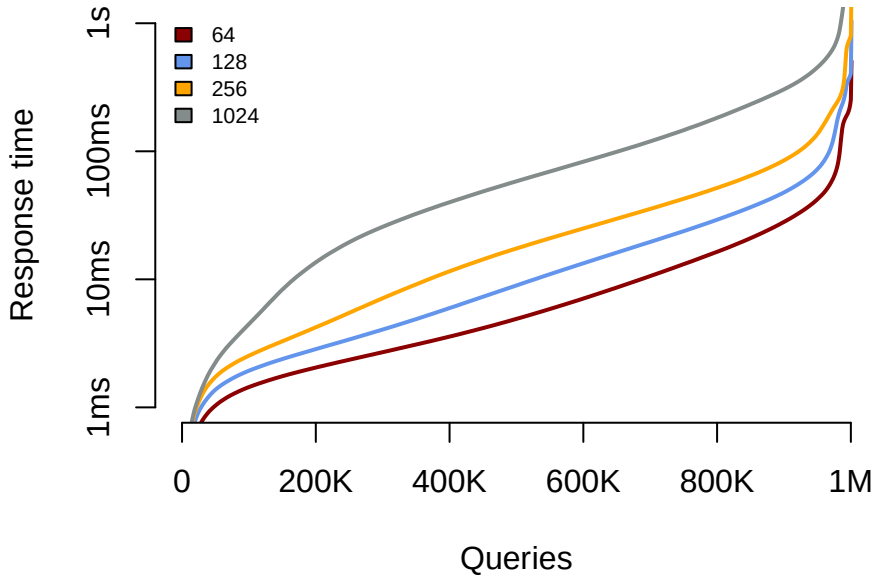

FIGURE 5.8: Latency distribution; 2M corpus; varying moq

moq	run time (s)	throughput (q/s)
64	204	4902
128	178	5618
256	157	6369
1024	134	7463

TABLE 5.1: Run time and throughput; varying moq

corpus	run time (s)	throughput (q/s)
0	30	33333
2M	204	4902
10M	757	1321
20M	1512	661

TABLE 5.2: Run time and throughput; varying corpus size

and 20M. Again, 1M user queries were replayed in each instance. The run time and throughput figures are shown in table 5.2.

These results do not show good scaling behavior in terms of corpus size: the decrease in throughput is significant when we grow the corpus. This, at least to a degree, is expected for the current implementation: larger intermediate result sets need more network bandwidth and processing time.

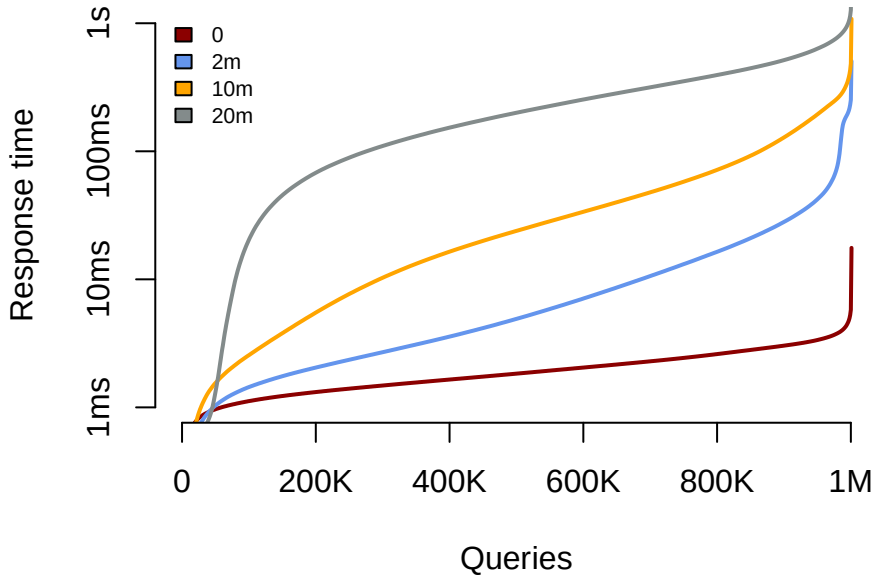

FIGURE 5.9: Latency distribution; varying corpus size; 64 moq

5.4 Comparative performance tests

To put the latency response distribution of `swarm` in perspective, we obtained a similar metric of the system currently in use by Spotify, and a distributed setup of ElasticSearch [Banon 2012].

We used an out-of-the-box ElasticSearch setup, where the index was distributed over 16 shards on the 8 available machines. Additionally, we used a replication factor of 2, resulting in each host serving 4 shards. For data, we used a track index of 10M documents (we did not use the full index mainly due to the fact that adding documents over the HTTP API interface is tediously slow on ElasticSearch - adding 10M documents took more than 3 days). The track index was generated by extracting the exact document entries (consisting of field-value tuples) that would normally have been inserted into the Spotify's service Lucene index. As a result, the documents ultimately stored in the Lucene index were consistent between the ElasticSearch setup and the Spotify service setup, and e.g. field-based searches yielded similar results.

The Spotify service was set up on a single host with 32G of available memory and 4 quadcore processors. We used a more or less vanilla version of the service, as used in the Spotify backend itself (and, in fact, temporarily repurpused a normal backend machine for these experiments). One modification we did was to prevent the service

from generating and presenting *did you mean?*-suggestions. The service used the full Spotify catalog, searched in all the (track, artist, album and playlist) indexes and filtered for correct geographic availability.

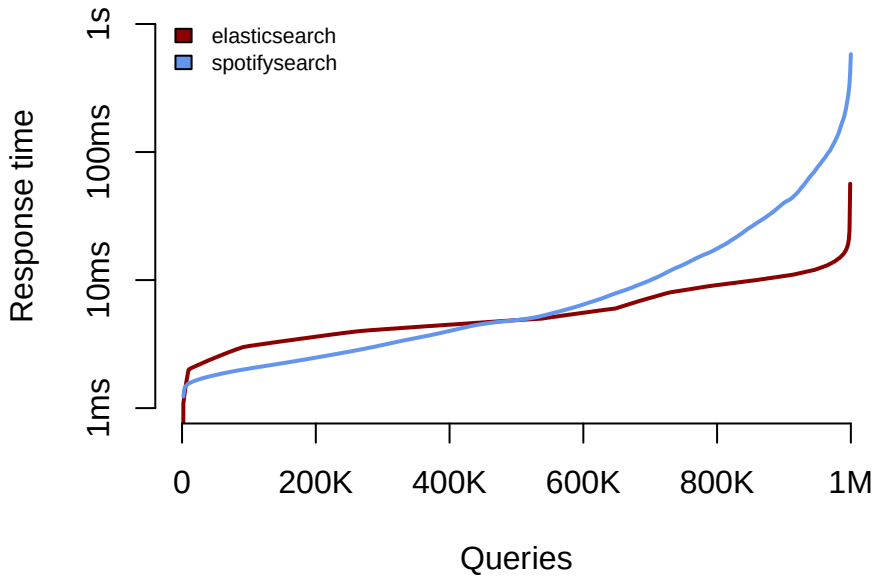

FIGURE 5.10: Latency distribution of ElasticSearch and SpotifySearch

Chapter 6

Conclusions

In this chapter, we summarize our analysis of the Spotify search service. We elaborate on the various database techniques used to implement `swarm`, our term distributed full text search index prototype. Additionally, we analyse its performance through interpretation of the findings of chapter 5.

6.1 Search at Spotify

We charted the Spotify track indexes and search logs (figures 3.3 through 3.6) to be able to reason about appropriate data primitives for the search system's use case. Comparing those figures to to similar systems in the domain of web search, we found that users use fewer terms in their search queries and (perhaps related) that queries are more repetitive.

6.2 swarm techniques

Several unconventional techniques in our prototype implementation, which were used in an attempt to overcome limitations of the term sharded search architecture, to improve performance or to support more expressive searching, are discussed below.

- **Index segmentation through forked read process.** Our approach of attaining rolling updates and separation of reading and writing processes around the `fork()` API was loosely based on a similar application in redis (see section 4.5). The experiments we performed do not allow us to quantitatively make assertions about this approach by itself (compared to more traditional approaches, for instance). We are confident, however, that it does not significantly impede write

or read performance. Our implementation is completely *lock-free* (but, of course, ultimately, locks are used in kernel space).

- **Enhanced pipeline model.** The proposed pipeline model allows for arbitrary complex tree expressions, including phrase searches, thereby improving on the implementation as suggested by Moffat e.a. [Moffat et al. 2007], who state that *word positional offsets are not stored.*

- **Inverted list subdistribution.** Based from the data from section 5.2, we assert that subdistributing inverted lists, as explained in section 4.7, contributes to favorable scaling characteristics and mitigates issues of unbalanced load, as experienced by Moffat e.a. [Moffat et al. 2007]. The cost of this approach is worse index writing performance (figures 4.6 and 4.7) and maintaining a shared state between all shards containing the term-distribution map.

6.3 swarm performance

We believe that `swarm`'s figures, as obtained in section 5.3, are within reasonable latency constraints, especially for the smaller corpus sizes and considering that these are obtained from a prototype implementations without an active query cache.

We are, however, not satisfied with the scaling characteristics exibited by these experiments: a practical application for any such system would be for significantly larger corpus sizes (e.g. 500M documents, when it would outgrow a single system's main memory). In the next section, we elaborate on some ideas for future research that could mitigate some factors that we believe contribute to this behavior.

Interpreting the comparitive results of `swarm` versus those of the current, full-fledged Spotify system (i.e. supporting different indexes, geographic filters), as done in section 5.4, we see no absolute benefits of `swarm` yet. Also, ElasticSearch seems to perform superior. This is as one would expect with the maturity of both full text indexing systems taken into account.

In the end, though, we only established the emperical performance difference to hold for the current dataset size. Ultimately, term-distributed search system should have favorable scaling characteristics. The main conclusion we draw here is that `swarm` does not perform unreasonable or unusable for a dataset of several million documents, despite some of the unmitigated scaling issues.

6.4 Future work

First, we regret that some individual features we implemented for our prototype system were not tested quantatively. For instance, a quantitative figure indicating the effect on both read and write performance of the *forked read process* approach, remains unclear. A comparive performance evaluation for different *sublist split thresholds* was not performed. The specific influence of our *list subdistribution* feature on query performance is unknown.

Secondly, since the ability of term distributed systems to double as a distributed cache (as explained in section 1.1.2), we had initially planned for our `swarm` prototype to include such functionality. Based on figure 3.4 and our results in figure 5.9, we still suspect implementing a distributed cache would significantly improve query latency and throughput figures, especially in the use domain of Spotify. Such a cache functionality could be implemented with various degrees of complexity: we suspect a very simple LRU cache for complete queries would already be highly effective. Interesting to see would be if more sophisticated approaches (e.g. that allow partial cache hits) would yield even more significant wins.

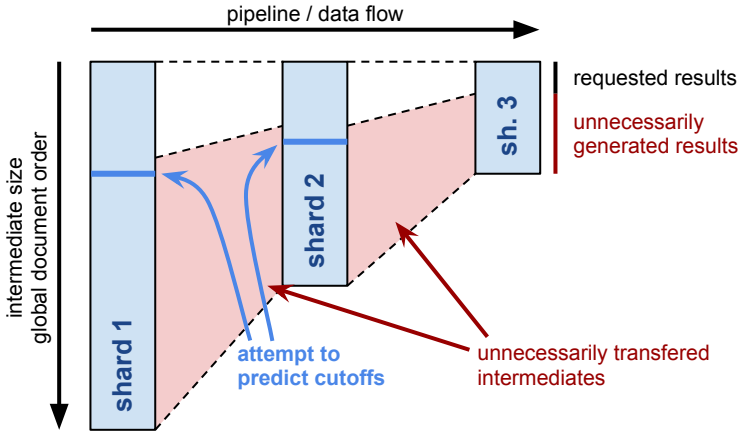

FIGURE 6.1: Cutoff prediction of query pipeline

Slightly related is the complexity of the query planner of our prototype. The heuristic currently used to estimate inverted list sizes (as explained in section 4.4.1) is coarse, and it would be interesting to see what improvement could be achieved if the planner had more detailed *cardinality* statistics.

We wonder how `swarm`'s non-ideal scaling behavior in terms of *corpus* size could be mitigated. The linear relation between corpus size and the size of intermediate result sets (that need to be transferred over the network) seems like a challenging problem to overcome. One idea we currently have is to predict safe cutoffs at earlier nodes in the pipeline based on term cardinalities. This approach is depicted in figure 6.1: since the

document are ordered by some (linearly distributed) global significance factor, we could try to guess the appropriate cutoff of this factor. We imagine experiments that test different heuristics to assign such cutoffs and (through replaying a query log) determine how many result sets fall short results versus the amount of network data transferred.

Aside from more experimentation, we firmly believe that a lot of small tweaks could significantly improve the system's performance, both in terms of implementation (e.g. perform more advanced profiling) and design (e.g. add features like Scholer's list delta compression [Scholer et al. 2002]). To support continuation of this research, we are currently putting effort into releasing `swarm` under an open source license.

BRIN, S. & PAGE, L. The anatomy of a large-scale hypertextual Web search engine, Computer networks and ISDN systems 30 (1998), 107–117.

LAMPE, C., ELLISON, N., & STEINFIELD, C. A Face (book) in the crowd: Social searching vs. social browsing in Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work, (2006), 167–170

MANNING, C.D., RAGHAVAN, P., & SCHUTZE, H. Introduction to information retrieval, Cambridge University Press Cambridge(2008)

MOFFAT, A., WEBBER, W., ZOBEL, J., & BAEZA-YATES, R. A pipelined architecture for distributed text query evaluation, Information Retrieval 10 (2007), 205–231.

RIBEIRO-NETO, B.A. & BARBOSA, R.A. Query performance for tightly coupled distributed digital libraries in Proceedings of the third ACM conference on Digital libraries, (1998), 182–190

BANON, S. elasticsearch (2012) http://elasticsearch.org/

DEAN, J. Building Software Systems at Google and Lessons Learned (2010) http://static.googleusercont
DL-Nov-2010.pdf

JEONG, B.S. & OMIECINSKI, E. Inverted file partitioning schemes in multiple disk systems, Parallel and Distributed Systems, IEEE Transactions on 6 (1995), 142–153.

TOMASIC, A. & GARCIA-MOLINA, H. Performance of inverted indices in shared-nothing distributed text document information retrieval systems in Parallel and Distributed Information Systems, 1993., Proceedings of the Second International Conference on, (1993), 8–17

BAEZA-YATES, R. & CAMBAZOGLU, B. Distributed Web Crawling, Indexing, and Search (2008)

SMILEY, D. & PUGH, E. Solr 1.4 Enterprise Search Server, Packt Publishing(2009)

PEREZ, S. indextank GitHub page (2012) https://github.com/linkedin/indextank-engine

HATCHER, E. & GOSPODNETIC, O. Lucene in action, Manning Publications Co.(2004)

KREITZ, G. & NIEMELA, F. Spotify–Large Scale, Low Latency, P2P Music-on-Demand Streaming in Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on, (2010), 1–10

SPOTIFY Spotify website: background information (2012) http://www.spotify.com/se/about-us/press/background-info/

BARBAY, J., LPEZ-ORTIZ, A., LU, T., & SALINGER, A. An experimental investigation of set intersection algorithms for text searching, Journal of Experimental Algorithmics (JEA) 14 (2009), 7.

BELL, T.C., MOFFAT, A., NEVILLMANNING, C.G., WITTEN, I.H., & ZOBEL, J. Data compression in fulltext retrieval systems, Journal of the American Society for Information Science 44 (1993), 508–531.

GRAY, J. & REUTER, A. Transaction processing: concepts and techniques, Morgan Kaufmann(1993)

MYSQL, A.B. MySQL (2005)

DOUGLAS, K. PostgreSQL, Sams(2005)

SCHWARTZ, B., ZAITSEV, P., TKACHENKO, V., ZAWODNY, J.D., LENTZ, A., & BALLING, D.J. High performance mysql, O'Reilly Media, Inc.(2008)

CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W.C., WALLACH, D.A., BURROWS, M., CHANDRA, T., FIKES, A., & GRUBER, R.E. Bigtable: A distributed storage system for structured data, ACM Transactions on Computer Systems (TOCS) 26 (2008), 4.

LAKSHMAN, A. & MALIK, P. Cassandra: a decentralized structured storage system, ACM SIGOPS Operating Systems Review 44 (2010), 35–40.

ELLIS, J. Leveled Compaction in Cassandra (2011) http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra

DEAN, J. & GHEMAWAT, S. LevelDB project page (2012) http://leveldb.googlecode.com

BAKER, J., BOND, C., CORBETT, J.C., FURMAN, J.J., KHORLIN, A., LARSON, J., LON, J.M., LI, Y., LLOYD, A., & YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services in Proc. of CIDR, (2011), 223–234

KLOPHAUS, R. Riak Core: building distributed applications without shared state in ACM SIGPLAN Commercial Users of Functional Programming, (2010), 14

ESCRIVA, R., WONG, B., & SIRER, E.G. HyperDex: A Distributed, Searchable Key-Value Store for Cloud Computing, (2011),

KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., & LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web in Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, (1997), 654–663

SILVERSTEIN, C., MARAIS, H., HENZINGER, M., & MORICZ, M. Analysis of a very large web search engine query log in ACm SIGIR Forum, (1999), 6–12

IMATRIX ZeroMQ website (2012) http://www.zeromq.org/

SANFILIPPO, S. & NOORDHUIS, P. Redis http://redis.io

SCHOLER, F., WILLIAMS, H.E., YIANNIS, J., & ZOBEL, J. Compression of inverted indexes for fast query evaluation in Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval, (2002), 222–229