Managing Index Repartitioning

Njål Karevoll

Master of Science in Computer Science
Submission date: March 2011
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

In the coming years, cloud computing is slated to become even more wide-spread as more and more companies have shown interest in outsourcing the maintenance of the hardware of their computing systems. The ability to scale up and down the amount of available computing power by requesting and terminating instances on-demand is a powerful tool.

Computing in the cloud is usually associated with running costs where the customers pay by the hour. Being able to scale down to the fewest possible number of servers make economical sense. In order to handle periods of higher load, scaling up to use more servers is also important.

The service infrastructure has to be able to scale both up and down to benefit from these features. In a search system, this calls for the ability to support different partitioning schemes and being able to seamlessly migrate from one scheme to another.

This master thesis will address managing repartitioning, rebalancing, adding and removing index nodes in a search system without disrupting running services.

Assignment given: 02. November 2010
Supervisor: Svein Erik Bratsberg, IDI

Abstract

Careful architectural decisions are required in order to create a highly available and scalable search system. This requires an in-depth analysis and understanding of the architecture and context of each deployment. Different requirements placed upon the system by different deployments mean different solutions provide the best case by case result, thus benchmarks provide an invaluable source of information.

Deploying in the cloud creates a lot of opportunities that have not previously been available for smaller companies and organizations, such as requisitioning additional servers to handle periods of high load. Cloud computing is usually charged by the hour, and being able to scale down to the fewest possible number of servers over time makes economical sense. In order to handle periods of higher load, scaling up to use more servers is also important.

This thesis provides an overview of common components and important aspects of a distributed search system. It then gives an overview of different partitioning techniques before going into the details of repartitioning and rebalancing in a document-partitioned full-text search system.

The thesis introduces a processing framework that draws inspiration from flow-based programming, which is shown a valuable tool in creating custom tailored search solutions. The implementation is used to benchmark different repartitioning and rebalancing strategies that are important to support in order to scale both in and out.

In conclusion, the techniques mentioned in the thesis show great promise in creating custom, maintainable and flexible partitions. The processing framework enables each specific deployment to easily compare different partitioning schemes and associated manageability and maintenance costs to determine the best fit for any given situation.

Preface

This master thesis was written for the Department of Computer and Information Science at the Norwegian University of Science and Technology under the supervision of Svein Erik Bratsberg with additional guidance by Simon Jonassen during the early stages.

I would also like to thank Alex Brasetvik and Heine Kolltveit for important discussions and feedback during the writing process.

Trondheim, March 29, 2011
Njal Karevoll

Contents

List of Figures

Chapter 1

Introduction

The focus of this thesis is on distributed search systems where the index is partitioned across one or more servers.

Economic motives may call for repartitioning since running the system on the fewest number of servers possible is cost-saving. In most a cloud environments [1] computing power is paid for by the hour, thus being able to scale down is an important property in order to keep the fewest number of nodes running over time. However, handling peaks in the load is important, which can be solved by starting one or more temporary servers.

Partitioning, splitting a large set of data into multiple smaller sets, is often used to increase performance with operations that can be performed concurrently on multiple partitions.

Repartitioning means adjusting the number of partitions and is required in order to support scaling up and scaling down. *Rebalancing* is ensuring the documents are distributed evenly between the nodes in a system, and that every document is in the correct partition.

Administrating fewer servers is intrinsically easier, but a distributed system can provide higher availability, better resilience to failures and in some cases better performance. This is a tradeoff between manageability, performance and availability.

Changes in usage patterns, both from moving from a testing environment to a real-world deployment and from changes in user behavior may significantly change the requirements of the partitioning. User behavior is closely related to, but not limited to, the query response time, recall and precision [2]. As these properties improves, users will use the search more and the load on the search system increases.

This master thesis presents a distributed search system implemented by solving the distribution as a data flow problem. By combining a processing framework based on well-known flow-based programming principles and existing full-text index servers as storage backends [3][4], the result is a distributed and scalable search system.

This master thesis is organized as follows. Chapter 2 gives an overview of existing systems. Chapter 3 introduces concepts and the components used in distributed search systems. Chapter 4 defines the scope of the thesis. Chapter 5 describes motivation for partitioning and introduces different partitioning strategies. Chapter 6 explains the distinction between repartitioning and rebalancing and discusses online and offline changes to partitioning schemes. Chapter 7 presents a flexible framework that assists developers and search implementers in quickly adapting to the changing requirements and varying load of real-world search systems. The implementation is used to benchmark the strategies introduced in this thesis in Chapter 8. Finally, conclusions and ideas for future work are given in Chapter 9.

Chapter 2

Prior work

This chapter gives an overview over the existing partitioning and repartitioning support in currently available search-related software.

2.1 Partitioning in full-text indexes

The two largest and most well-known open source full-text indexes, Solr [4] and Xapian [3] include support for distributed querying, but the index partition construction and maintenance is left for the users of these systems to implement. This has opened up the market for third parties to provide components to support distributed indexing, some of which are mentioned in Section 2.3.

2.2 Partitioning in distributed databases

In recent years, many distributed database systems have been created. These are often built with distribution in mind and many of them include support for adding and removing partitions.

MongoDB [5] is a document-oriented database that supports automatic partitioning by applying a dynamic range-partitioner to partition the input data into chunks. Rebalancing (called balancing by the MongoDB documentation) is done by using a two-phase commit protocol [6] to exchange ownership of the data chunks between the partitions.

Redis [7] is a key-value store that divides the keyspace into a set number of hash slots, which can be assigned to nodes. Data can be migrated between the nodes at any time.

Many similar solutions exist, such as CouchDB [8], Amazon Dynamo [9], Cassandra [10], Voldemort [11] and Memcached [12].

None of the currently available distributed databases include viable support for full-text indexing and querying. This means that developers have to choose between having the features of a full-text index or the ease of distribution, partitioning, repartitioning and rebalancing provided by the distributed databases.

2.3 Distributing full-text indexes

Adapting existing full-text indexes to use distributed databases and file systems as the backend store has been done by several software projects.

Katta [13] is a data storage layer that provides distributed storage for Lucene by splitting the data into shards that are assigned to server nodes. Katta uses Hadoop [14] to create the indexes and store them in a shared filesystem, either locally or remotely. This system is geared towards batch indexing, and continuous or real-time updates are not directly supported and has to be manually implemented.

Solrandra [15] uses Cassandra as a peer-to-peer database to store the inverted index. In this distributed system the entire index is available in any participating node in the database ring [16]. The backend indexes can grow quite large, giving diminished returns of the query node caches because the caches become less likely to contain the required data.

2.4 Comparison

The available solutions presented in the previous section are generally tightly integrated with their respective full-text index servers and data access libraries, which may be undesirable from a support standpoint.

In contrast to the implementations described in the previous sections, which are implemented by integrating into existing full-text indexes, the distributed search system described in the following chapters operate on as a layer on top of the existing full-text indexes. In this system, updating to the partitioning functions (see Chapter 6) to accommodate changes in the partitions becomes trivial operations that only involves updating a single processor or pipeline (Chapter 7) in a data flow graph (Section 3.3).

As shown in Figure 2.1, this data flow processing framework can be used to provide additional features on top of existing full-text indexes, custom index adaptations or even full-stack frameworks that uses partitioning or distribution internally.

In this thesis, this method are shown to be flexible, scalable and easily maintainable.

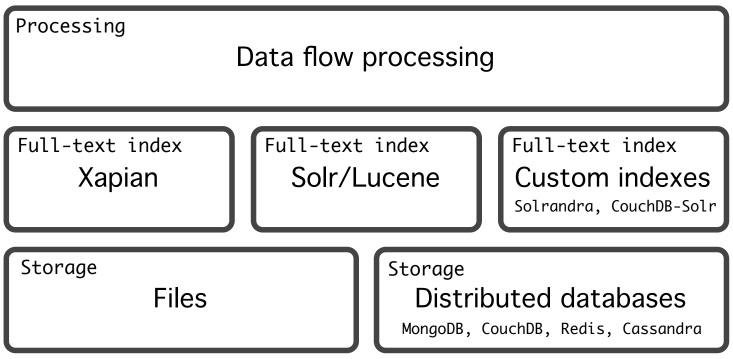


Figure 2.1: Overview and comparison of some existing systems.

Chapter 3

Components

This section describes some common components and important aspects of a distributed search system.

3.1 B+trees

The B-tree is a sorted tree data structure [17] that is similar to the binary tree [18] except it allows more than two paths out from a node which is essential to the implementation of dynamic databases [19]. The tree is always balanced in the sense that all the leaf nodes are at the same distance from the root node.

B-trees are used in full-text indexes because it is a data structure that scales well and provide fast lookups and updates, with only some overhead when nodes have to be split.

A B+tree is a variation of a B-tree that stores the data only at the leaf nodes. Additionally, each data block has a pointer to the next data block in order to perform sequential scans.

3.1.1 Operational complexities

Lookup requires one operation for each node for a total of $m = \log_{n+1} N+1$ where $n+1$ is the order of the tree and N is the total number of nodes, which gives $\mathcal{O}\left(\log n\right)$.

Inserting requires lookup up the correct node, writing the data block and rebalancing the tree, which gives $\mathcal{O}\left(\log n\right)$.

Deleting requires looking up the node, performing the deletion and rebalancing the tree, which gives $\mathcal{O}\left(\log n\right)$.

More detailed analysis of the amortized running times can be found in [20].

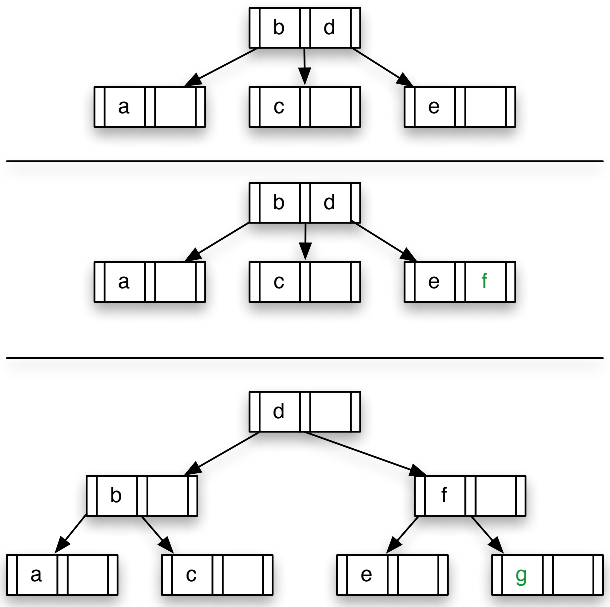


Figure 3.1: Adding the letters “f” and “g” to a 2 B-tree

3.2 Full-text index

The system described in this paper assumes that a full-text index is used. We use Xapian [3] by default, but its usage is loosely coupled and can easily be replaced with other full-text indexes providing the same or similar services, for example Lucene [21], Solr [4] or Terrier [22].

Operating on documents on this level isolates the indexing processes from the details of the full text indexing service used which helps ensure the versatility and pluggability of the system. It also means that new full text index implementations can quickly be tested in order to perform benchmarks, analyze performance or comply with special deployment-specific requirements.

A full-text index is usually stored in multiple tables, implemented as B-trees. To better understand how production quality full text indexes work and to be able to reason about operation costs, we look at the tables [23] used in Xapian:

Position - contains the positional information for terms which is used for phrase searches.

Termlist - stores the list of terms for each document.

Postlist - contains summary statistics for the index, metadata, document lengths, document values and the posting list.

Record	- contains the document data, as set by `document.set_data()`.
Spelling	- contains data used for suggesting spelling corrections.

Not all deployments will use all of the tables. For example, the position table may be empty in indexes where query features such as phrase searching or nearness will not be used, in order to save space. The position table is often the largest table by quite a large margin due to its need to store multiple locations for each term.

The record table may be empty if the document data is not required or is stored in another database, the latter requiring a mapping to exist between the xapian document ids and the database that contains the database data.

3.3 Flow-based programming

Flow-based programming [24] is an application development methodology that uses a network of usually asynchronous, reusable components that operate on and transforms streams of data. Many of the concepts are found in older computer science literature from [25] and have recently gained in popularity as multiprocessing on multi-core systems has become even more common.

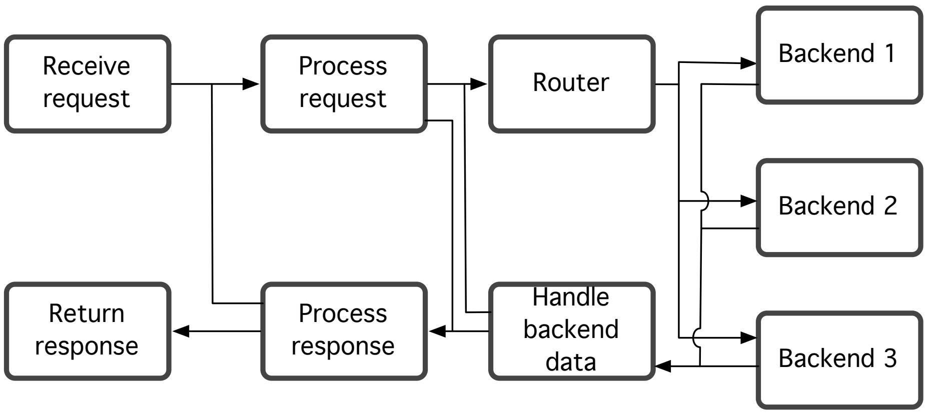


Figure 3.2: Flow based programming.

The data flows are usually processed asynchronously, which means it is not possible to accurately predict the time it will take to process any single packet. However, it is possible to configure a mode of operation by configuring the local execution models, the network and communication protocols that ensures timeouts or guaranteed packet acknowledgements can be relied on in most use cases. This is a flexibility that allows any component to reuse other components or even networks.

This methodology has seen much use in application domains such as banking, sales and scientific applications dealing with continuous streams of data, which makes it a very interesting base to build upon. Recent commercial activity around this methodology from larger vendors are Yahoos pipes [26], IBMs Damia [27], MuleSoft [28] and Apache Camel [29].

3.4 Nodes

A node is a single process that has contains one or more pipelines and communicates with other nodes via the network.

3.5 Message queues

A Message Queue is a component that is used for communication to decouple the sender and the receiver of messages. It provides an asynchronous communication channel between one and more senders and zero or more recipients.

One of the most widely known protocols used by messaging systems is called AMPQ (Advanced Message Queuing Protocol) [30], which is a vendor-neutral protocol for communication between a messaging provider and a client. The client in this sense is any sender or any receiver. Messaging then acts as a piece of middleware that enables each service that communicates with it to be unaware of the other services since they only need to know the network address of the middleware. The middleware takes care of routing messages to the correct recipients based either on the queue used or any routing data associated with the message. If the message is undeliverable, the middleware may store the message for delivery at a later time when the appropriate client connects.

Different messaging systems support different policies which includes, but are not limited to the following:

Reliability - what delivery guarantee exists on queued messages, and whether the sender is notified of dropped messages.

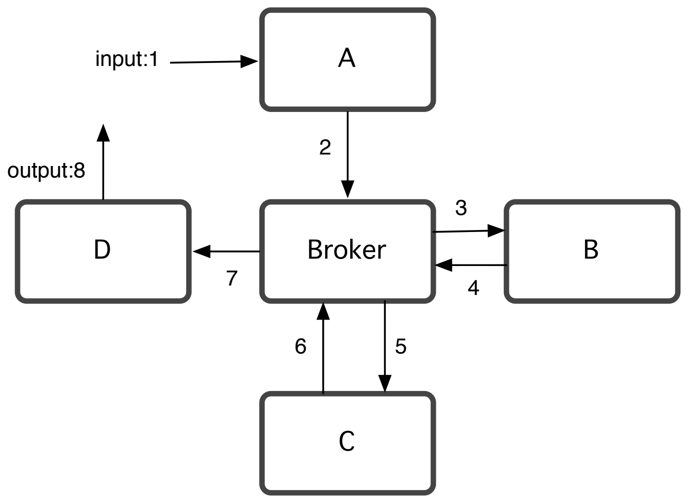
Order of message delivery - are the messages delivered in the same order they are sent?

Durability - how the provider handles situations where the intended recipient is temporarily available and what happens when the broker is restarted.

Notifications - whether the system allows for notification of message delivery and other clients sending/receiving messages.

There are many message queue systems, some of the most well-known being RabbitMQ [31], ActiveMQ [32], ZeroMQ [33], Amazon SQS [34].

Out of the aforementioned message queue systems, only ZeroMQ supports broker-less [35] operation, which eliminates the broker as a central bottleneck in a messaging network (see Figure 3.3), giving lower latency and higher throughput at the cost of reduced manageability. The manageability may be regained by using distributed brokers and directory services.



The same system, but without the broker uses significantly less queues, which translates to fewer messages being sent.

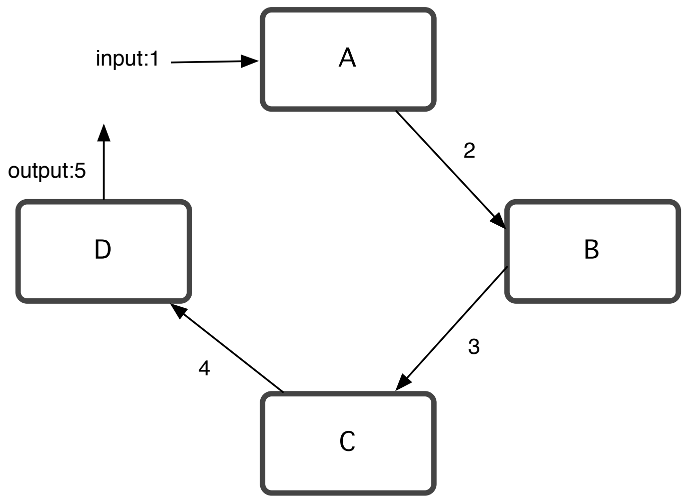


Figure 3.3: Broker versus broker-less message queue systems.

3.5.1 Messaging patterns

Message queue services enable system architects to use different message patterns. The most common messaging patterns are request-reply, publish-subscribe and push-pull.

Request-reply messaging

Request-reply messaging is similar to the traditional client-server model where a client sends a request and a server responds with a reply.

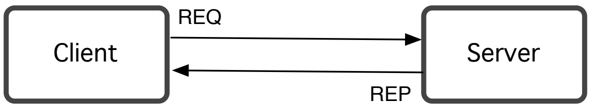


Figure 3.4: The request-reply messaging pattern.

The request-reply pairs may be *lockstep* or not. Being lockstep means that a client cannot have more than one outstanding unanswered request at any time. The requests are usually load-balanced between available servers and servers usually use fair-queuing to make sure that no single client starves the other clients.

Publish-subscribe messaging

In publish-subscribe messaging, messages from publishers are delivered to subscribers according to the message contents, often a prefix. Multiple subscribers may receive the same message if they are subscribing to the same type of messages, and publish-subscribe messaging can be said to be a broadcasting messaging system.

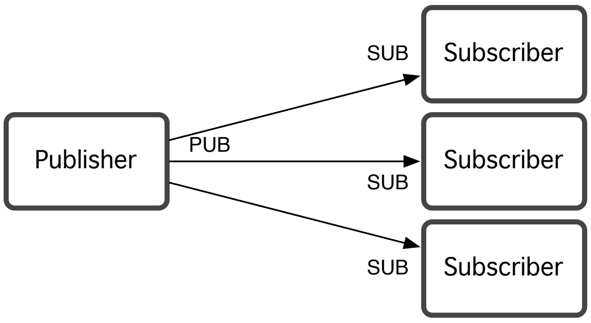


Figure 3.5: The publish-subscribe messaging pattern.

Push-pull messaging

In push-pull messaging, messages are load-balanced between available clients. The sender is called a pusher because it actively sends the data to clients and the clients are called pullers because they have to actively register with a server in order to be eligible for receiving data.

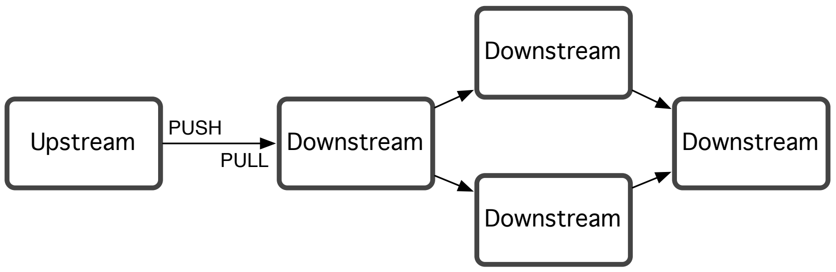


Figure 3.6: The push-pull messaging pattern. The nodes are labeled upstream/downstream as seen from the leftmost connection.

Push-pull messaging is also referred to as pipeline messaging because multiple push-pull patterns may be used serially to arrange a set of nodes in a pipeline. Data always flows in the same direction.

3.5.2 Message serialization

Messaging layers usually only support sending bytes, and nodes operate on structured data. In order to send structured data between nodes, the data needs to be serialized before sending.

JSON [36] is an excellent candidate for this serialization requirement due to its low overhead and ease of use. Bindings for JSON exist in most programming languages which makes it easier to integrate services written in different programming languages.

Other significant candidates are Protocol Buffers [37], MessagePack [38] and BSON [39]. Each candidate has its strength and weaknesses with certain types of data, and it is up to the implementors to decide on a standard for each deployment.

A low overhead measured in message sizes is important in a distributed network as the bandwidth may be limited. For some of the messages may be possible to save some bandwidth by compressing the objects after serializing it, trading CPU cycles for network bandwidth [40]. In some cases, it may be reasonable to batch multiple documents together before compressing them to share the compression overhead. This technique becomes better if the messages that are batched together and compressed share characteristics that enable effective compression [41].

An important thing to note with compression is that depending on the message size, it may consume a significant amount of memory on either side of the message transmission as the compression and decompression uses memory to store parts of both the uncompressed and the compressed messages at the same time. It is possible to peek at the compression statistics to determine the exact sizes of the uncompressed messages and attempt to make sure that enough memory for the operation is available in addition to using stream-based compression.

A small-scale comparison is shown in Figure 3.7. 10000 documents were randomly selected from Wikipedia. Their average size were measured before they were run through a text-processing pipeline (see Section 7.3) that performed some basic textual analysis and created a document ready for insertion into a full text index. The document were serialized with each of the serializers and their new average message sizes were measured.

Format	**Before text processing**	**After text processing**
bson	1249.61	3238.26
json	1264.50	4404.47
msgpack	1232.60	3226.97
pickle	1305.95	5827.16

Figure 3.7: Serialized document sizes, before and after text processing.

JSONs poor performance in this test can be slightly attributed to the fact that it was unable to encode the binary data that was the serialized full-text index document, which had to be base64-encoded in order to get encapsulate the conflicting bytes from conflicting with the JSON serialization.

3.6 Networks

A network is any number of interconnected nodes that work together in order to provide a service.

Nodes may be connected to each other with different kinds of connections, and everything from direct connections using raw sockets to high level messaging using perspective broker [42] or a messaging queue system is possible.

A simple network, shown in Figure 3.8 may contain several nodes, message queues and support different protocols:

Sources pushes data to any available text processors and subscribes to messages from the index in order to know when queued documents have been indexed.

Text processors pulls data from a source, processes the data and pushes the data to an index.

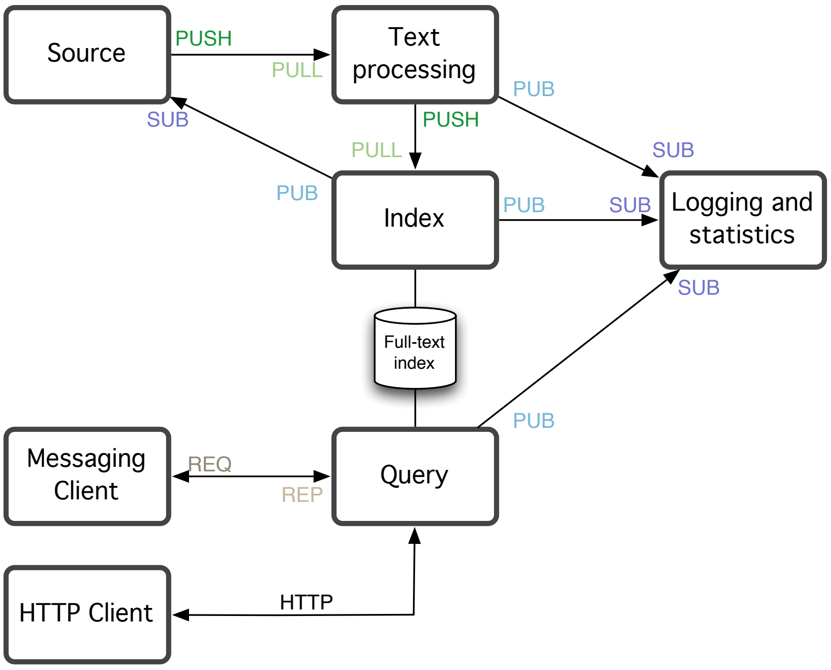


Figure 3.8: A simple network of nodes that are connected using message queues.

Indexes pulls processed document and writes them to an index.

Clients sends requests to query nodes using different protocols.

Query nodes responds to requests from clients.

Logging and statistics can be gathered by subscribing to updates from different services in a network.

More durable and complicated networks would usually include directory services that acts as a lookup service in order to determine message destinations, and distributable message brokers using load balancing.

Chapter 4

Limitations

This chapter describes the limitations of this paper.

The processes and operations defined in this paper operate on documents and does not perform optimizations that might have been possible if it was implemented at a lower granularity level such as term-partitioning. This is due to term partitioning requiring a global lexicon, which does not scale well [43], and even though it has lower utilization of resources like disk accesses and transfers a smaller amount of data volume per query document partitioning is the scheme usually chosen by web search engines [44], and will be examined in more detail in Section 5 and 6.

4.1 Single writer and multiple readers

This is the way both Xapian and Solr/Lucene work and sets the limitation that only one thread may modify the database at any time. Multiple threads and processes are allowed to read the database at any time, however.

Because of this, scaling up indexing in a search system practically requires partitioning to be implemented.

Since we are building on top of these full-text indexing systems, these limitations are imposed on the indexing nodes.

4.2 Fault-tolerance and error recovery

The design and architecture of a fault-tolerant system is out of the scope for this paper, but due to the important of fault-tolerance for applications like search systems that operate on arbitrary data, we will have a short look at what it means.

A key concept in software fault-tolerance is to hierarchically decompose large systems into modules, each module being a unit of service and a unit of failure. Module failures should not propagate beyond the module

[45]. Processes may achieve fault containment by sharing no state with other processes; making its only contact with other processes via messages carried by a messaging system.

Non-transient errors such as bad sectors on hard drives, memory corruption and the like are also not considered in this paper.

4.3 Byzantine nodes

Byzantine [46] or other purposefully dysfunctional nodes are not considered and are beyond the scope of this paper as they are unlikely in a controlled environment.

4.4 Replication

This paper assumes replication as a layer between the indexes and the query nodes and we will not deal with the details of how this replication is done, but we will deal with the configuration and updates to the replication layer as a part of repartitioning. As such, it is important to know a couple of properties we assume holds for the replication.

In practice, `xapian-replicate-{client,server}` can be used to replicate databases from a master database which modifications are made on, and a set of secondary read-only slave databases to which these modifications are propagated [47], shown in Figure 4.1. Similar services exist for other full-text indexes such as Solr [48].

For example, to support a high query load there may be many query nodes, each with a local copy of the database, and a single indexing node. In order to allow scaling to a large number of query nodes with large databases and frequent updates, we need an database replication implementation to have the following characteristics:

The replication protocols such as the one in Xapian [49] supports efficient replication frequent small updates to large databases without interrupting searching on slave databases and indexing on the master database during synchronization. It does not invalidate up-to-date data cached in-memory on the slaves.

It worth noting that this replication in Xapian does not support replication and aggregation of multiple writable databases to a single database.

4.5 Resource bounds

No checks are performed on the resource usages, such as checking that there is enough free disk space available to repartition and rebalance.

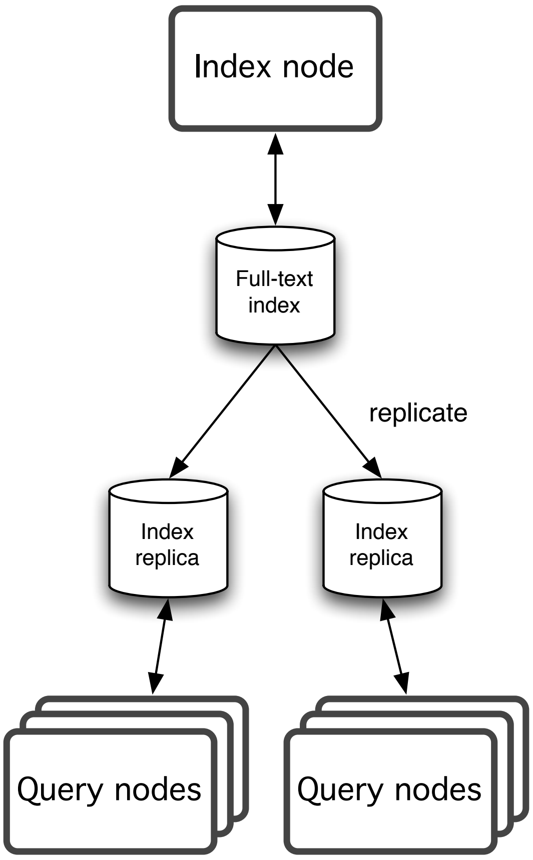


Figure 4.1: Example replication of a full-text index.

Chapter 5

Partitioning

The focus of this paper will be on partitioning in an environment as defined in chapter 3.

Partitioning means dividing the data up in n number of independent subsets and is a tool that is central to scaling systems horizontally, and is usually employed to achieve the following properties:

Manageability – managing smaller amounts of data is intrinsically easier even though the partitioning introduces some complexity itself

Performance – running an algorithm on a data set size n is the size of the data, can be sped up linearly by partitioning the data into m partitions and running the algorithm in parallel. This also opens up the possibility for keeping the dataset in memory in each partition and allows for better use of multi-core and multi-node systems.

Availability – a single point of failure is undesirable for systems that favor high availability. With multiple servers providing the same service, the system can still attempt to provide the same functionality even though one or more of the partitions are unavailable. This results in a lower mean time between failures in the system, since the chance of any of the nodes having an issue that stop from performing properly is higher than it would be if there only was a single node. This is due to the fact that given an individual failure chance of p_e in a single server situation, the chance of being in a degraded state is p_e^n in a multi-server solution assuming serial failures. The remedy for this is replicating the nodes.

Document partitioned systems (Figure 5.1) have n self-sufficient indexes, each containing a partition of the document collection. The total size of the partitions are slightly larger than the non-partitioned case, as some of the data occurs in multiple partitions. For further discussion about this overhead, see Section 5.7.

this	1,2,3,4
is	2,3,4
a	1,2,3
test	3,4

Using document partitioning to partition this index into two partitions can yield the following two indexes:

this	1,2
is	2
a	1,2

this	3,4
is	3,4
a	3
test	3,4

Figure 5.1: An unpartitioned index and a document partitioned index.

Each partition maintains its own inverted file for the *documents* it contains. In order to rank queries, this requires each node to be synchronized with the rest of the nodes with respect to the dictionary statistics, unless the partitions are of such size and composition that they can be assumed to encompass the necessary properties of the collection as a whole.

5.1 Improving query throughput

From the perspective of a read-only query-service, the use of partitioning is usually either to load-balance or load-divide. Load-dividing between partitions aims at making the fewest amount of partitions required perform the necessary work to answer a request. Load-balancing by partitioning tries to achieve an equal average load between the partitions and usually requires an additional merge-step, as the results from each partition has to be merged to form the complete result.

Partitioning can also be used to allow queries to be performed on only specific subsets of the complete data set, which may dramatically reduce the required computation time. The prerequisite is being able to perform some kind of collection selection, which puts some limitations on the partitioning method used, which we discuss further in Section 5.4.

If every or even just the most heavily-used parts of the index can fit in memory on a node, it will be able to handle the requests much quicker than a larger node that has to use (more) disk accesses to do the same.

Some partitioning strategies (discussed in more detail in Section 5.4) enable the use of collection selection [50] at query-time to select one or more candidate partitions that may contain relevant results.

5.2 Improving index throughput

Partitioning enables us to scale up the write-performance of a system by parallelizing the write operations.

A server in the cloud can use partitioning locally to improve the indexing throughput by attaching multiple volumes and partition between them instead of only using a single large volume. Doing this increases the write-bandwidth of the server and requires a system that natively supports this kind of partitioning. This partitioning can be hidden from the rest of the system by providing a query service that merges the results from querying these partitions.

Indexing nodes are under the limitations of only allowing a single thread to write (see Section 4.1) to a database at a time. Most modern computers have a CPU with multiple cores. In these systems, partitioning may also become important if the indexing process is CPU-bound instead of only being relevant if the indexing is I/O-bound.

5.3 Avoiding partitioning

Partitioning are seen by some [51] as often being a sign of premature optimization, that it is relatively hard and complicates deployments. In many cases this may be correct and is supported by Moore's law [52]. Many capacity problems that have been predicted for the future has not come to pass due to this effect.

But Moore's law may not hold forever: even though it fits the trend of the past 50 years, it does not necessarily mean it will continue to hold for all eternity. It also does not cover I/O latency and bandwidth, even though SSD disks have significantly improved random reads and writes [53], which are frequent in search systems.

Depending on future improvements and reduced costs of hardware to address scaling may be risky. Especially if customer growth is taken into account, which is multiplicative to the regular growth of just the existing customers.

If the initial scale of the system is too large for a single machine to handle, then partitioning is a hard requirement.

Additionally, in the cloud [1] it is easier to handle variable load and scaling by using several smaller, more transient nodes in order to scale than relying on the latest and greatest hardware to provide the necessary computing resources. In fact, most cloud service providers do not expose any hardware to the users, which means that the size and computing power of most of the virtual servers are significantly lower than what is commercially available for managed hosting providers.

Attempting to prepare for deploying on multiple servers by partitioning on a single physical disk drive is an example of premature optimization that actually prevents a system from achieving optimal performance. If two partitions are configured on the same physical disk drive, heavy indexing or querying can cause the processes to interrupt each other, which is easiest to see in disk usage (Figure 5.2).

Full-text indexes such as Xapian write out parts of the document data to disk during indexing, even when not committing, which results in the physical disk having to seek. This is generally a good idea, as it saves a lot of time writing the documents to disk when committing. This improves the global system resource usage by spreading the I/O operations required to commit the documents over time, improving the total service throughput [40].

However, if two processes both experience the same high indexing load, disk thrashing could occur.

This disk thrashing does not occur with small databases, but as the databases grows, the document data being written to disk require more and more disk seeks due to the growing B-tree tables.

5.4 Strategies

In this section we will discuss some different partitioning strategies, partitioning functions and their attributes. The difference in these attributes and the domain we are indexing determines which strategy is the best fit for each specific deployment.

A partitioning function is a function used by a partitioner to select an output partition (Figure 5.3) for a given partitioning key (see Section 5.4.1)

5.4.1 Partitioning key

A partitioning key is the value the partitioning functions operate on in order to determine an output partition. This key is arbitrarily defined by the system configuration and can either be a *static* or *computed*.

Static keys are keys that occur directly in document fields. Examples of these are titles, tags, modification timestamps or other content.

Computed keys are keys that may be computed by taking the document fields as input. Examples of these are content hashes, combinations of

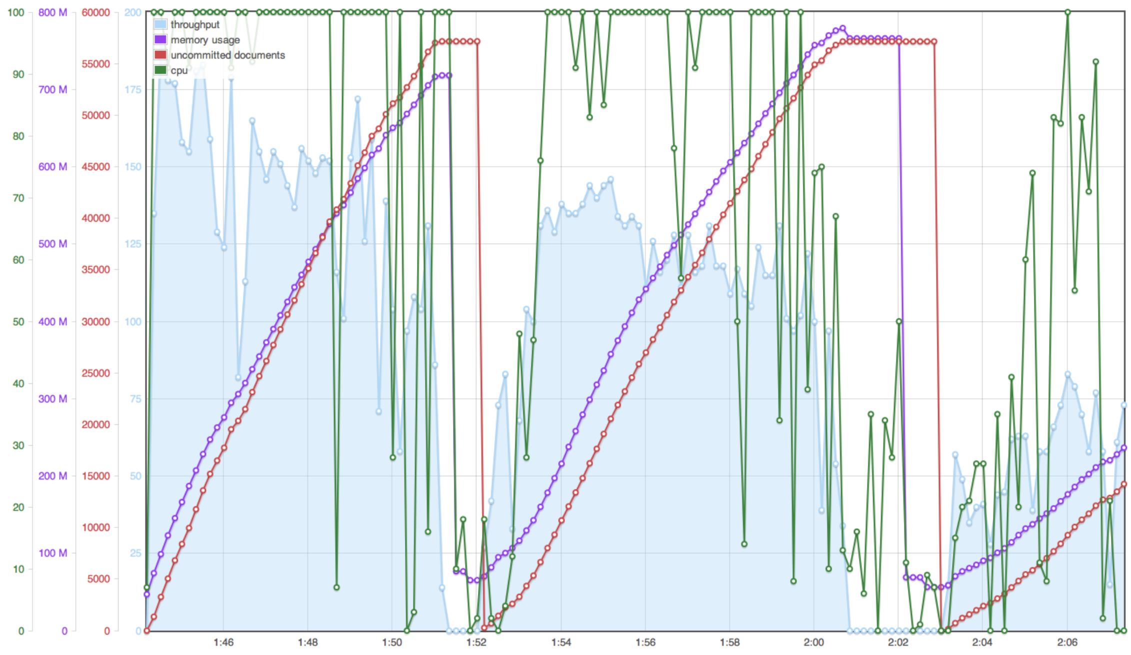


Figure 5.2: An example that shows disk thrashing that severely limits the throughput of an indexing service. After the second commit, the throughput (light blue) is less than half of what it originally was, due to disk seeks occurring during indexing.

modification timestamps and titles and so on.

Both static and computed keys rely on operating on fields that are present in all the documents.

We define a *repeatable* partitioning function as a function that always yields the same partition for repeated calls with the same arguments. This allows for *in-place* updates and deletes.

In-place updates and deletes refer to being able to send updates and deletes to the same node that currently has the indexed document, and assists in lowering the amount of consistency anomalies which is discussed in further detail in Section 6.4.

5.4.2 Hash partitioning

A hash function is a function that transforms it input into a shorter representation that serves as a signature for the original input [54].

Hash partitioning is done by using a hash function on the partitioning key in order to select a partition for input documents. The output from the hash function is passed through an additional modulo operation where the modulus is the number of partitions.

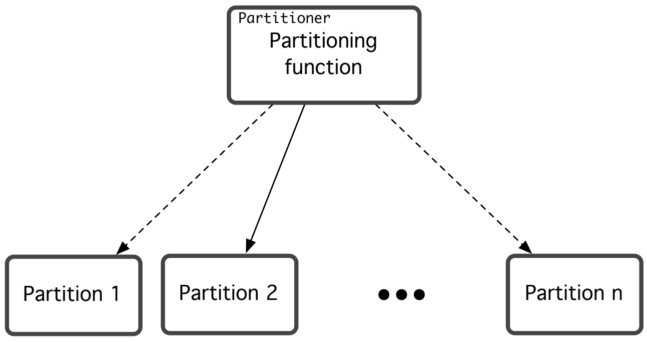


Figure 5.3: Partitioning function used by a partitioner

This partitioning function gives different partitioning attributes depending on the value being hashed. If a unique identifier (see Section 5.6.1) is used as the partitioning key, all partitions must be queried in order to answer a query, but each partition are likely to have fewer disk reads in order to respond to the query. If some sort of category attribute or tags are hashed, it is possible to use the contents of the query in order to select a minimum number of partitions that are guaranteed to contain the only relevant categories or tags.

In either case, hash partitioning is a *repeatable* partitioning strategy.

5.4.3 Round-robin partitioning

A round robin partitioning ensures that each partition receives document its document n before any partition receives its document $n+1$ in a circular order.

Round-robin partitioning can be accomplished by using fair-queuing [55], a feature provided by most messaging systems. This provides a uniform distribution of documents over the partitions, but is not repeatable.

A great advantage with partitioning strategy is that it does not necessarily require a separate partitioning service.

5.4.4 Random partitioning

A random partitioning ensures that there is no correlation between the document and the selected partition.

This type of partitioning does not guarantee a uniform distribution, and is not repeatable.

Random partitioning can be accomplished in a number of ways. Pseudo-random partitioning using load balancing is the fastest, but does not guarantee a uniform distribution. This type of partitioning is not repeatable.

5.4.5 Load balanced partitioning

In load-balanced partitioning, each incoming document is sent to the partition that has the lowest load at the time (see Figure 5.4). This results in the highest partitioning and indexing throughput, as each partition is allowed to operate at its maximum capacity.

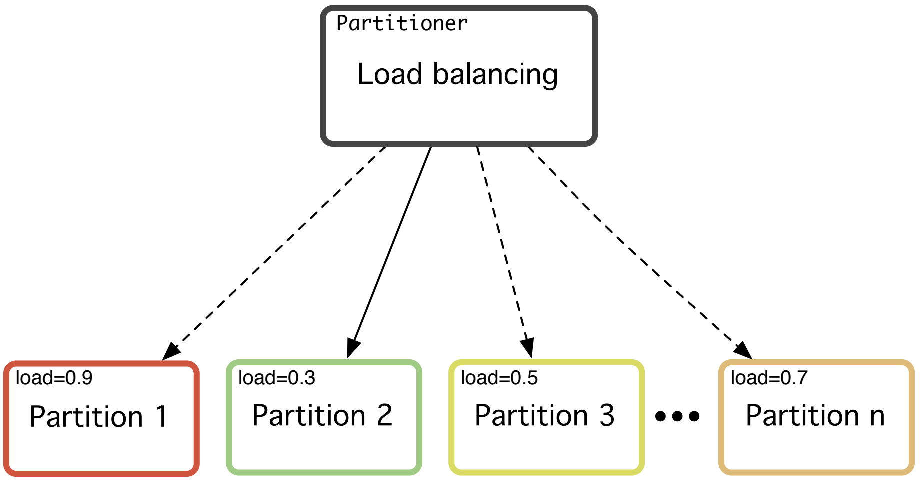


Figure 5.4: A load balancing partitioner always selects the partition that is currently experiencing the least amount of load.

A uniform distribution when using load balancing will only be achieved if each of the partitions has the same capacity and the documents are of the approximate same sizes.

This partitioning strategy is not repeatable.

5.4.6 Range partitioning

In range partitioning, a set of input ranges are either configured or computed. During partitioning, the partitioning key is matched against these ranges, and the range that contains the key determines which partition the document should be sent to.

Using this technique, the partitioner needs to know the value range of the partitioning key.

This strategy may be uniform if the ranges are selected carefully, and is repeatable.

5.4.7 Spatial partitioning

Spatial (or geo-) partitioning [56] means partitioning a document into a partition depending on its location in a document-space (see Figure 5.5). The uniformness of a spatial partitioning function depends on its exact definition, but is repeatable as long as it isn't *adaptive*.

Adaptive spatial partitioning uses already seen documents in order to adjust the borders in the document-space to create a more uniform document distribution.

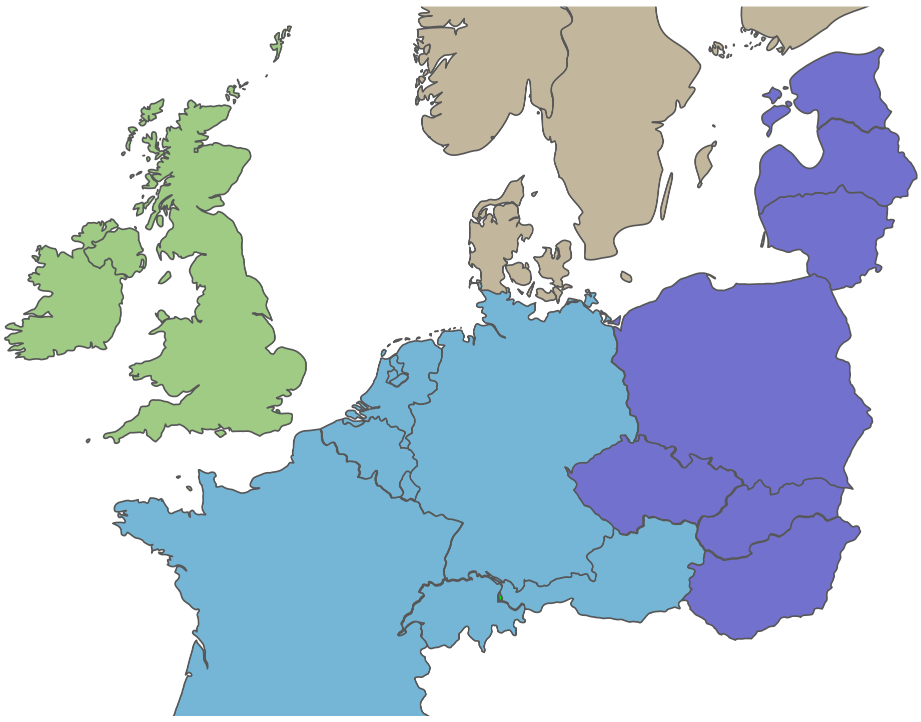


Figure 5.5: A spatial partitioning of some European countries.

5.4.8 Clustering-based partitioning

In general terms, a cluster is a set of entities that are alike [57], and entities from different clusters are not alike (see Figure 5.6). This is a natural fit when it comes to partitioning document collections since it implies that if one document from a collection is relevant to the query, the rest which is topically alike will have a higher degree of relevancy. Whether it is plausible to get enough information to direct a given query to a partition based on only the query string or if additional information is required (for example a user-specified cluster) remains to be seen in practice.

Clustering as a document partitioning strategy requires a good set of training documents that is used to generate the initial clusters. After the required clusters have been created, the documents can be put into an cluster by finding which of the existing clusters it is most similar to. Using clustering does not require any assumptions about the data that is common to most of the statistical methods for partitioning the data [58].

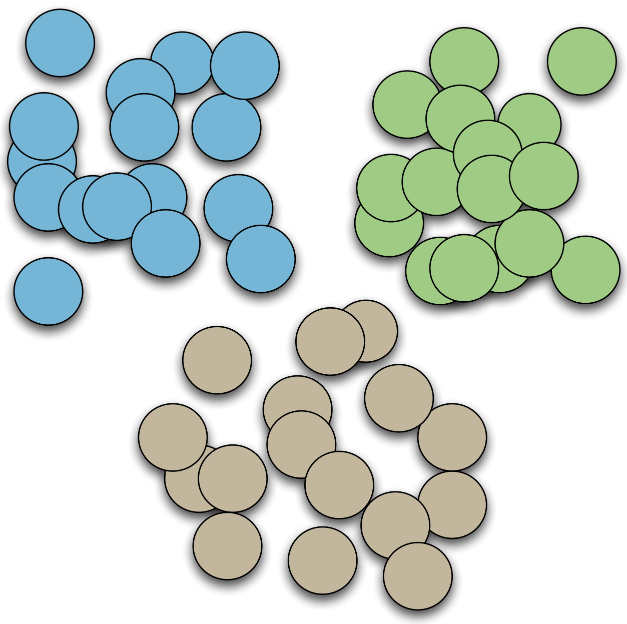


Figure 5.6: A set of documents partitioned into three clusters based on similarity in two dimensions.

Some popular methods of clustering includes the following:

Hierarchical clustering [59] is done by recursively merging clusters, starting with a cluster size of one (*agglomerative* clustering), or by recursively splitting up a cluster into smaller clusters, starting with a single cluster containing all the data points (*divisive* clustering).

k-means clustering [60] is done by deciding the number of clusters k, seeding them somewhere in the data space and evolving them by continuously adding the data point closest to the center each of the clusters and recomputing the resulting new centers.

Both clustering and categorization are attempts to partition the dataset based on topics. Many other techniques have been presented and discussed [50], but it is still uncertain which partitioning scheme is suitable under which circumstances [43].

Partitioning based on clustering is repeatable as long as no re-training is performed.

5.4.9 Combinational partitioning

Combinations of different partitioning functions is possible by either using multiple partitioning functions in a single partitioning service (Figure 5.7), or combining multiple partitioning services (Figure 5.8). Using multiple partitioning functions in one service results the lowest amount of nodes and messages being sent, but using multiple services allows for more granular updates of the partitioning network without affecting services upstream in the network.

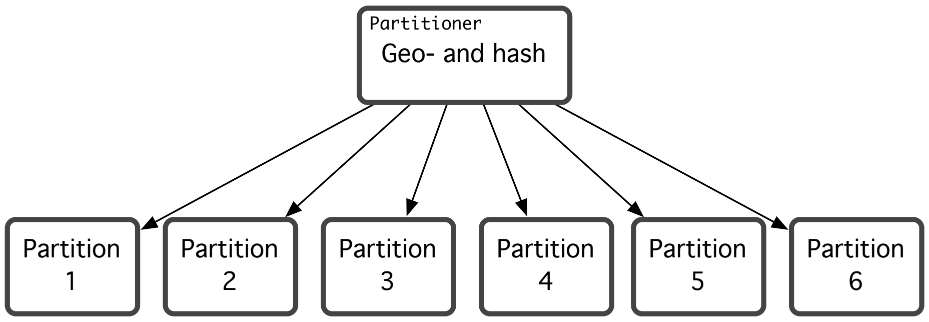


Figure 5.7: Combinatorial partitioning with multiple partitioning functions within a single service.

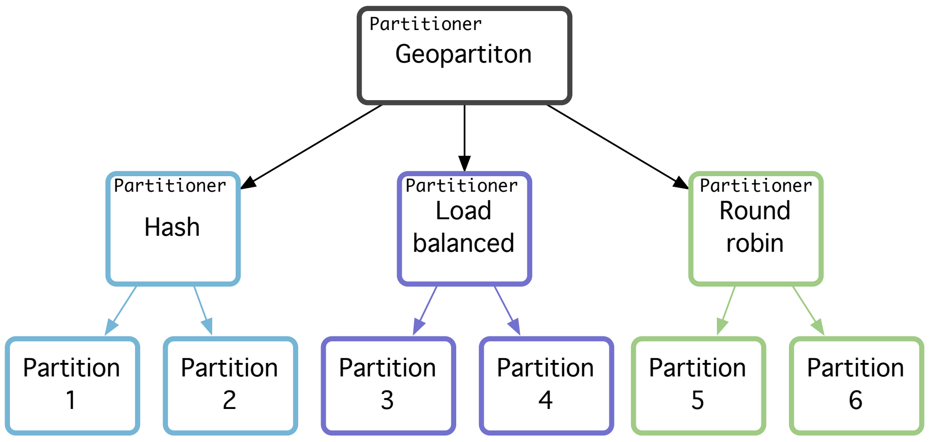


Figure 5.8: Combinatorial partitioning with multiple services each using a single partitioning function.

For example, if we were indexing companies and locations the data could first be partitioned by continent (and optionally country), and finally by some sort of hash-partitioning.

5.5 Collection selection

In order to query multiple index partitions for relevant documents, it is possible to simply broadcast the queries to all the partitions and merge the results. However, this could in some cases be costly, impractical or even unnecessary.

Collection selection can be considered the reciprocal of index partitioning during query-time (see Figure 5.9). The main point of collection selection is avoiding querying indexes that are unlikely to hold documents of interest to the user, and there are several possible methodologies that can be used.

D'Souza, Thom and Zobel [61] concludes that managed collections are likely to be used in distributed retrieval, and allows for more effective selection.

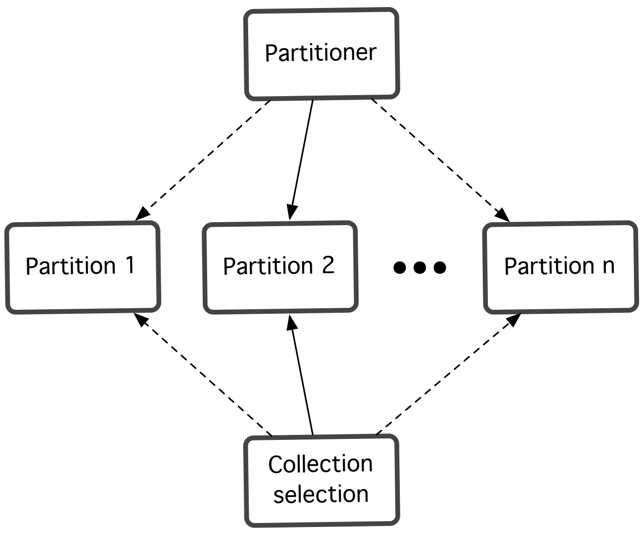


Figure 5.9: Using collection selection to limit the query to only use indexes that contain relevant documents.

5.6 Practical considerations

A partitioning on the document source or type can be used to create a facade interface to a search engine that manages multiple collections that share some common processing steps.

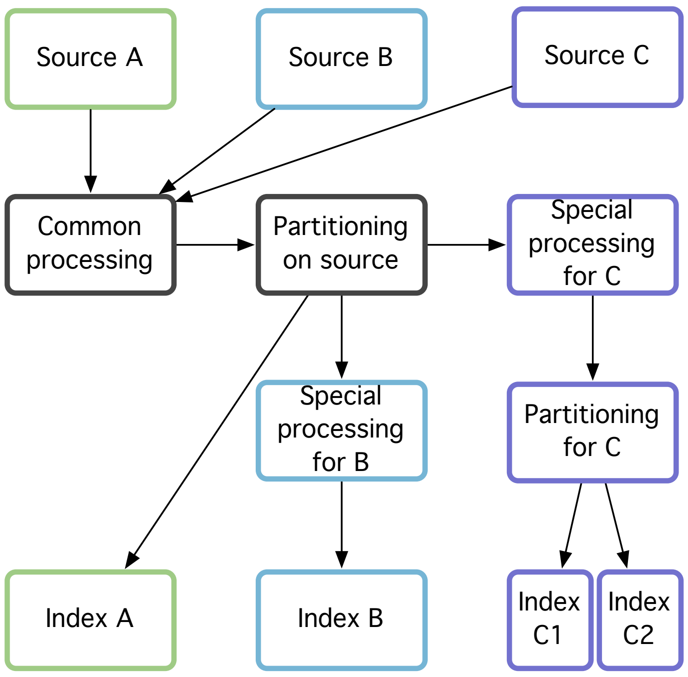


Figure 5.10: A partitioning service used to create a facade interface for multiple collections.

In Figure 5.10, documents from three sources (A, B and C) share some common processing steps. The nodes that perform the common processing sends all the documents to the same partitioning service, which splits the documents into three partitions, based on their source or intended destination index. A is then indexed directly, B has some extra processing done before being indexed, and C has both some more processing and an additional partitioning done before being indexed.

Neither partitioning service in the example needs to know anything about the processing that is done in its input or output networks.

5.6.1 Unique identifiers

Each document is assumed to have a unique identifier, which is an arbitrary term [62][63]. This can be anything from the url for web documents to SHA-1 hashes for git commits. Numerical identifiers or hashes should generally be avoided since multiple sources may index documents to the same index, and collisions, which are much more likely with such identifiers, would result in documents being overwritten. A solution to the problem is to prefix the numerical identifiers with a string that identifies the source. Due to the maximum term length requirements, short prefixes are generally better.

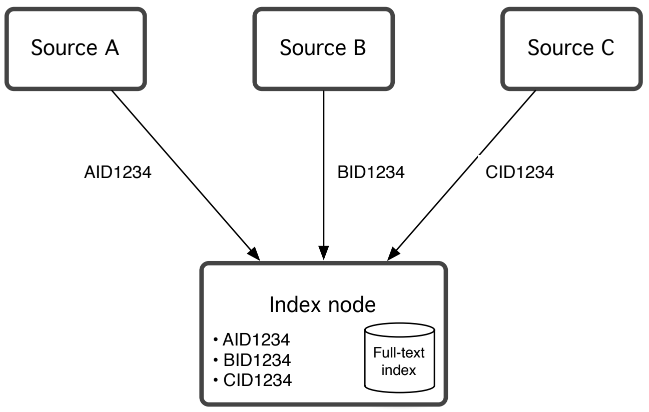


Figure 5.11: Prefixing IDs

The unique term must never occur in any of the other documents in the index, because if collisions occurs, there is no way to tell the documents apart, and the new documents would delete and replace the previously indexed documents.

A simple way ensure that a unique key does not naturally occur in any other documents is to keep all document terms in lowercase and using an uppercase prefix when constructing the unique term. A unique-identifier for a document with the numeric identifier *id* from a source *S* could then become "`SIDid`" (see Figure 5.11).

5.7 Size increase

Multiple smaller partitions takes up more hard drive space than a single large partition because two factors: the structural overhead posed by having more B-trees and the duplication of terms.

The latter factor overshadows the first, as parts of both the spelling and the postlist will be duplicated between the partitions. For partitioning strategies that does not consider the vocabulary used in the documents, this can become a significant overhead when compared to a lesser partitioned solution.

The following listing gives an idea of the size differences:

```
$ xapian-compact wex{1,2,3,4,5,6,7,8}.xapian wex_all.xapian
postlist: Reduced by 50% 106216K (210856K -> 104640K)
record: Reduced by 4% 3968K (84112K -> 80144K)
termlist: Reduced by 27% 36704K (133104K -> 96400K)
position: INCREASED by 3% 19272K (511872K -> 531144K)
```

```
spelling: doesnt exist
synonym: doesnt exist
```

5.8 Querying in a partitioned system

There are three architecturally different ways of querying in a distributed search system: *direct*, using an *implicit broker* or using an *explicit broker*. The following sections discusses these in further detail.

5.8.1 Direct access

The query node may access the index partitions directly, either via backend support, a networked filesystem (NFS [64]) or using a distributed filesystem such GlusterFS [65] or Hadoop DFS [66].

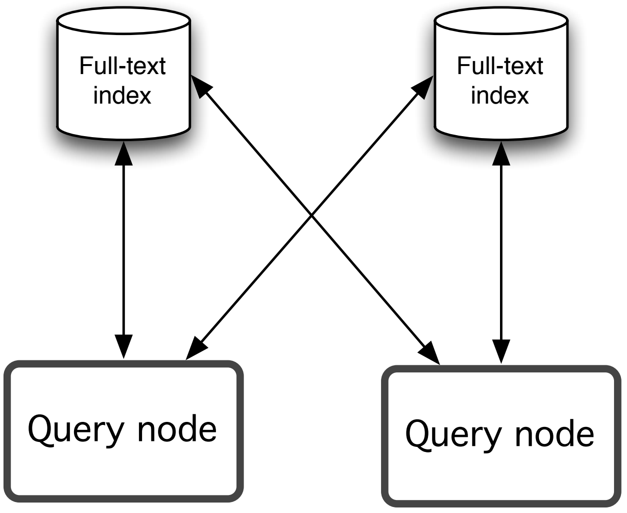


Figure 5.12: Querying without the use of a broker by accessing multiple indexes directly.

For example, the remote database feature in Xapian [67] enables querying multiple indexes, both local and remote, as if they were just a single database (see Figure 5.12). This requires an `xapian-tcpsrv` or `xapian-progsrv` instance running for each of the remote databases, and neither the protocol nor the sockets used between the client and the servers have any fault tolerance support. The remote databases does not have full support for spellings and metadata, which potentially could be used to implement core features such as spelling correction.

5.8.2 Implicit broker

In this approach, users send their queries to any of the query nodes, which will broadcast the query to the other query nodes before it starts perform the query on its own index. When all the results are available, the node responsible for performing the query on behalf of the user merges the them into a single response and sends it to the client.

Each query node has its own index and establishes connections with the other query nodes as required (see Figure 5.13).

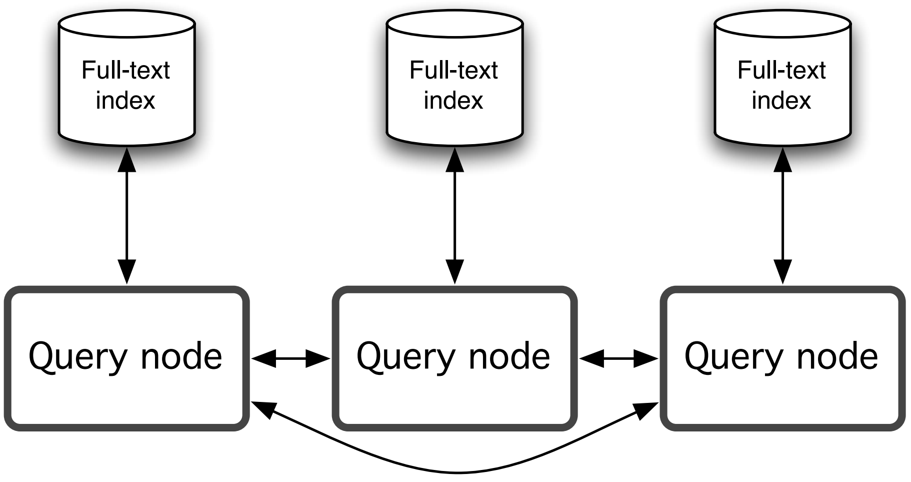


Figure 5.13: Querying by sending the request to any of the query nodes, which will then act as a implicit broker and query its siblings as part of its query processing pipeline.

If the query nodes have limitations on the amount of incoming connections or uses threads to perform the queries, this method is prone to *distributed deadlocks*. A distributed deadlock in the simplest case is when two nodes are waiting for a resource owned by the other node, such as a two-node system that only accepts a single incoming connection simultaneously gets a request that requires them to fetch results from the other node.

This is how distributed querying is done in Solr [68].

5.8.3 Explicit broker

This approach uses a separate *query broker* that manages multiple query nodes that perform the queries separately. The results from the query nodes are then merged as a separate step in the processing pipeline of the query broker (shown in Figure 5.14). This is a trade-off between the easy setup of the remote database feature and the control, customization and features offered by the explicit merging.

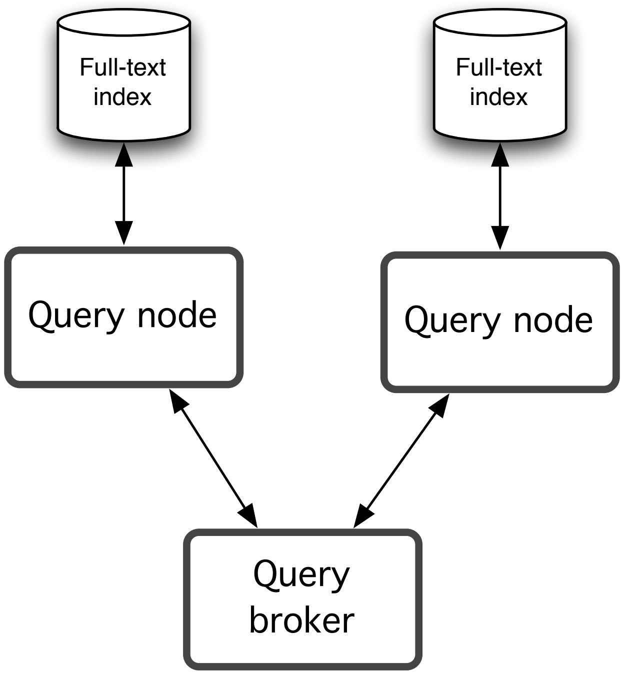


Figure 5.14: Querying with the use of a broker that merges the results from multiple query nodes.

Chapter 6

Repartitioning and rebalancing

Search systems are often used to index steadily growing amounts of data, and sudden, large and unforeseen changes to the data volume indexed by a system is not a very likely scenario. To accommodate the growing data volume we may choose to increase the number of partitions, and to save costs we may decrease the number of partitions, both of which is a process called “repartitioning”.

Administrators may want to reduce the number of partitions or servers used due to financial concerns or simply that the available computing power, memory et cetera have been outpacing the data growth. Cloud computing providers typically charge by the hour based on predefined server sizes, and being able to scale down an instance size or reduce the number of servers may be desirable from both an economical and maintenance perspective.

The number of clients in a search system may fluctuate considerably over time. If clients are only consumers, it is most likely enough to just add more query replications. However, if clients also produce data that are indexed either temporarily or permanently, it may be useful to be able to vary the number of partitions over time.

During early stages of deployment or testing, it may be desirable to test different partitioning schemes to find the one with the most desirable properties. Later in the system lifecycle, the administrator may find out that the original partitioning scheme has some undesirable properties and has to be replaced. For example, a system may have been implemented by partitioning by the zip-codes of the documents, but during the lifecycle, more accurate GPS tags have become available and we wish to partition on these instead. Being able to support changing the partitioning function completely without having to fall back to refetching the documents from the source can be very valuable, as refetching may be an expensive and time consuming operation. The data origin may also be a system that is under heavy load to begin with, and avoiding any further load may be crucial. Additionally, it is faster than a reindex because it does not have to go through any filtering or text-processing between the indexes.

A different argument for having to repartition is having several spikes on one partition while one or more other partitions has low load. If the spikes are consistent over time, it is a sign that the partitioning function used is suboptimal and should be updated. This may not be obvious before the service has been deployed and is getting real usage patterns that may not have been available in the testing stages.

Repartitioning is the act of changing the number of active partitions. Rebalancing is an optional step that is used to balance the amount of data between the active partitions, shown in Figure 6.1.

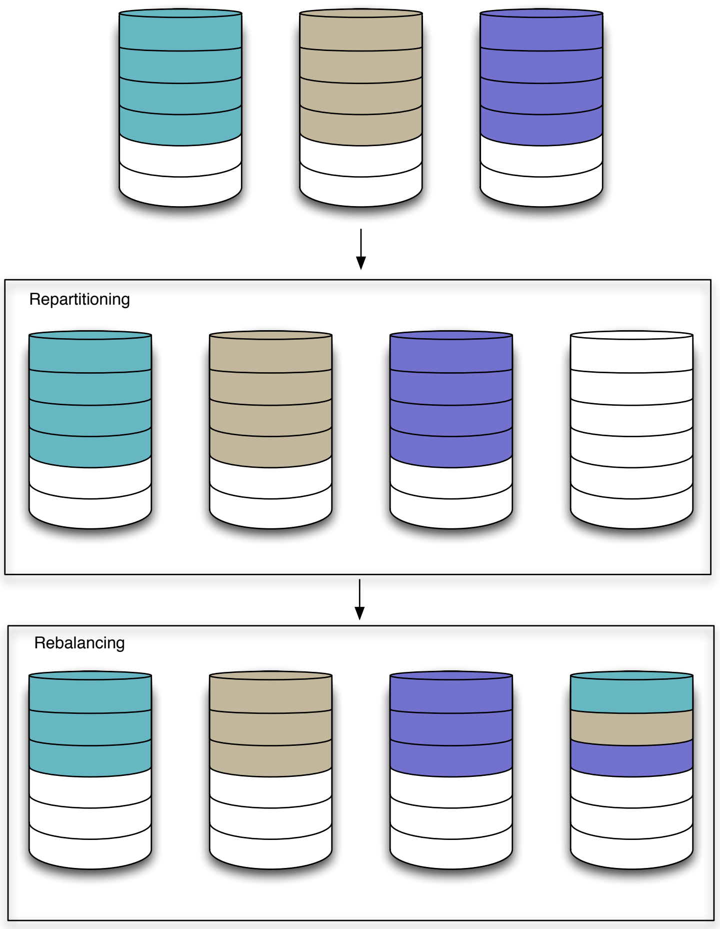


Figure 6.1: Repartitioning and rebalancing

6.1 Moving documents

Moving a document from one partition to another requires looking up the document parts in the full-text index tables, constructing a serialized document object from this data and eventually remove the entries from the tables (see Section 6.4 for more discussion on this topic).

The termlist and record tables are trivial to update, as they are keyed on the document identifier, while the position and postlist tables require multiple lookups in order to delete. Metadata associated with a document is, when stored keyed on the unique term, a single B-tree lookup and update to edit.

If a large amount of documents are moved to or from an index, compacting may be considered. Compacting is the process of reading a database and writing it out to disk without leaving much room for future modifications. The reason this helps is that the tables are implemented using B+trees, which may gradually increase in size, leaving spare space on each of the levels. This compacting process can result in large space savings, enabling more of the database to fit in main memory and available caches.

6.2 Repartitioning strategies

We distinguish between *offline* and *online* repartitioning. In offline repartitioning the index is unavailable and any index or delete operations has to wait until the repartitioning process is completed. In online repartitioning the index stays available and documents may continue be indexed or deleted during the operation, albeit at a potentially slower rate than during normal operation.

If the query service uses replicas of the original full-text indexes instead of relying on direct connections to the original full-text indexes it can be kept running during the repartitioning completely independent of whether the repartitioning is performed offline or online. However, the query nodes are still subject to the issues discussed in Section 6.4.

6.2.1 Offline repartitioning

The main focus of this paper will be on online repartitioning, but in the interest of completeness we will mention a mildly intrusive offline strategy that works well in cloud computing.

Offline repartitioning strategies is the most intrusive class of repartitioning strategies, but it is also the fastest as it does not handle the regular indexing load in addition to the load generated by the repartitioning. For medium-scale deployments in the cloud, a good approach to offline repartitioning is to use one or more high-memory nodes to perform the necessary merges and splits. High memory nodes are used because they enable the

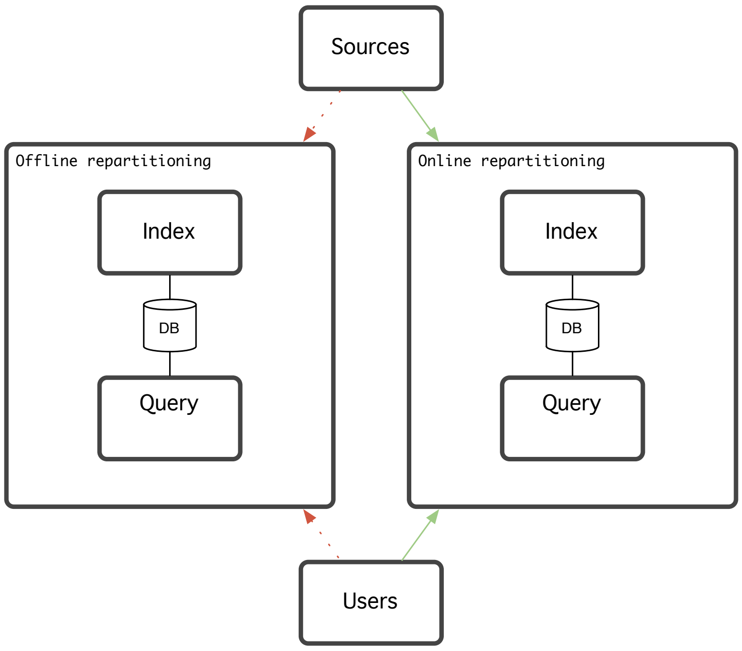


Figure 6.2: Offline vs online repartitioning

services to commit fewer times during the repartitioning, which saves a lot of time when working with large databases.

1. Bring down the indexing nodes and detach their volumes.
2. Attach a set of volumes to a high-memory node.
3. Perform the merges or splits and write the results on new volumes.
4. Create index nodes for any added partitions, if any.
5. Attach the new volumes to the indexing nodes.

This strategy uses two central attributes of cloud computing: temporarily using a high-performance server to speed up an operation and allocating or discarding volumes quickly to meet a variable storage demand.

The original volumes can be reused if the merger node only needs to do small-scale rebalancing, such as moving a few documents from one partition to another, or simply discarded if a larger operation, such as changing the partitioning key or changing the volume size is performed. As the total size of the documents affected by the repartitioning grows, reading from the original and writing to fresh volumes becomes faster as the reads and writes can be done in parallel.

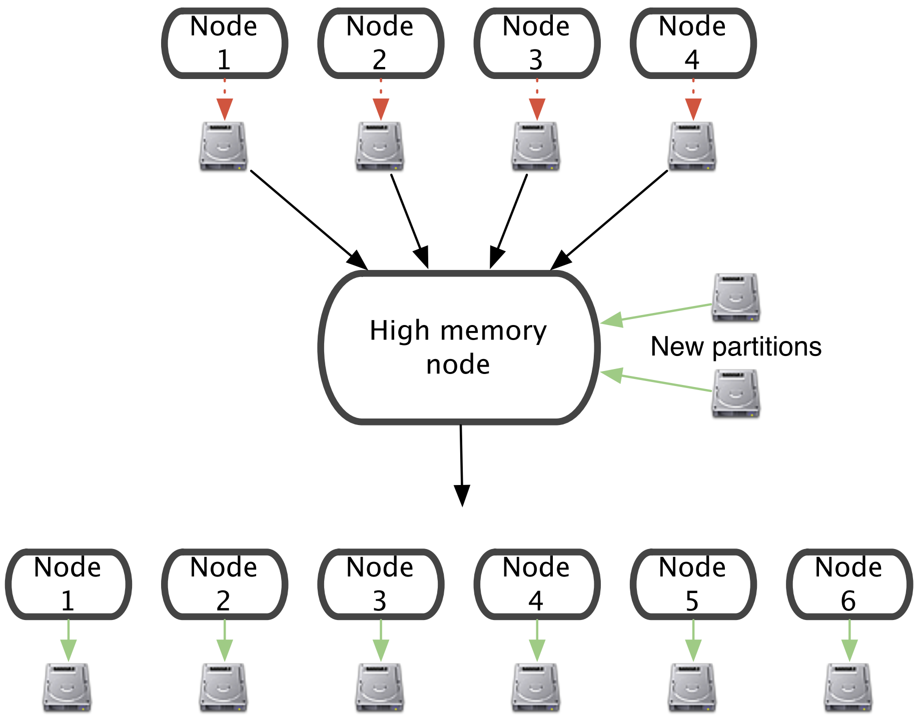


Figure 6.3: Offline repartitioning

Since cloud storage usage usually is charged by GB/month, allocating double the required data volume for a small period of time in order to save considerable time in repartition is a viable solution.

Optimization

The index nodes can be paused, flushed and have their full-text indexes detached in order to avoid having to shut them down completely. An index writer may be in the paused state indefinitely while the repartitioning is being performed. When the repartitioning is completed, the full-text indexes can be reattached to the index node and processing may resume. Changes in the number of index nodes however, still require changes to the networking graph, but not necessarily to any of the existing index nodes.

6.2.2 Online repartitioning

Using online repartitioning, the repartitioning of the full-text indexes is performed in a non-intrusive manner in order so other components in the search system can continue operating without noticing that any repartitioning is taking place. This class of strategies is still subject to the consistency issues discussed in Section 6.4.

Adding partitions

When increasing the number of partitions, the only changes required to the original network is the updating of the partitioner to include the added partitions. Some partitioning functions such as the spatial and clustering partitioning will require retraining while other partitioning functions such as load-balanced partitioning does not require any reconfiguring at all.

Rebalancing is optional depending on the consistency requirements (see Section 6.4) and the importance of quick offloading of the load on the original partitions.

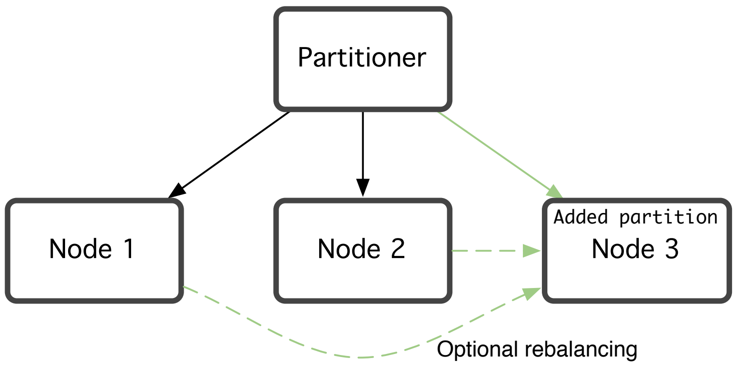


Figure 6.4: Online adding of a partition

Adding partitions in an online system will thus use the following steps:

1. Create the additional index nodes.
2. Set up replication between the new index nodes and query nodes if required.
3. Configure any query brokers to include the new partitions when executing queries.
4. Configure the partitioners to use the new partitions.
5. Deal with consistency requirements and rebalance if required.

Removing partitions

The changes required to the partitioner function is the same as in the previous section (6.2.2).

When reducing the number of partitions, rebalancing is not optional, as the removed partitions contains documents that would otherwise be unavailable.

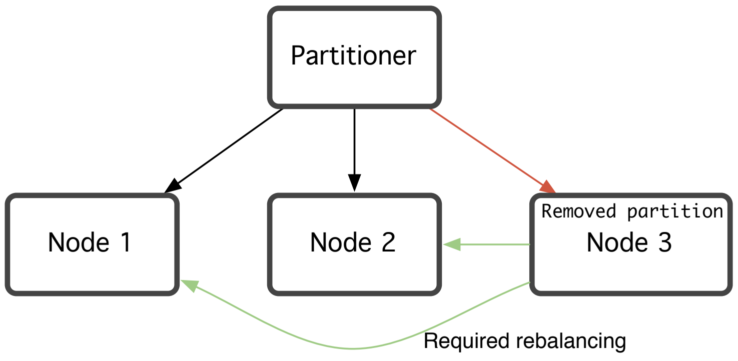


Figure 6.5: Online removing of a partition

Removing partitions in an online system is a two-step process:

1. Configure the partitioners to only use the remaining partitions.
2. Rebalance.

Handling frequent node changes

Whether the number of partitions are likely to increase again at a later time should be taken into account, as it might be feasible to just move partitions to the remaining nodes without performing any merging. This only requires configuring partitioners and replicas to reflect the new socket specification of the moved partitions.

This makes rebalancing the partitions trivial when increasing the number of nodes at a later time, as it will only require moving the partitions:

1. Move the partitions to the new index nodes.
2. Configure the partitioners to use the new nodes.
3. Configure the replicas.

The partitioner may be updated before the move is performed in order to ensure that new documents to the partition is sent its new location after the move. This utilizes the built-in queuing features of the message queuing service in order to queue the messages and deliver them to the correct node when it becomes available.

By doing this, no synchronization between a source and destination partition is required, but the moved partitions of the index will be unavailable during the move. However, the messages to the partitions will be queued, and will be sent when the partitions becomes available again. If spending memory on temporarily storing these messages is undesirable, swapping them to disk either by using the disk swapping features of the messaging system or using replaying (see Section 6.2.2) should be considered.

Moving a partition is a quick operation if it is performed by unmounting and mounting a SAN device [69] or using most networked file systems. If such services are not available, the partitions will have to be copied to the new node over the network. Transferring a complete full-text index over a network interface may take a considerable amount of time and in these situations freezing (see Section 6.2.2) may be used to keep the indexing process responsive and avoid excessive memory usage by messages piling up on the outbound queue of the partitioner.

Freezing

A different non-intrusive repartitioning strategy involved using a auxiliary "incoming"-index that incoming documents gets indexed to when the other indexes are down, as shown in Figure 6.6.

1. Route all incoming documents to a temporary "incoming" index which the query nodes replicate and use.

2. Freeze all the index writers, disconnecting the replicas.

3. Merge/split the required index writers.

4. Restart the replication process to the new index writers.

5. Route the incoming documents to the new writers.

6. Merge the incoming index to the new writers.

Freezing may be done either to a single auxiliary node or to many.

A single node may be able to handle the average indexing load of several original partitions as it starts with an empty database which eliminates much of the indexing overhead with regard to I/O from disk-seeks resulting from having to update large B-trees on physical hard drives.

If a single node is unable to handle the indexing load for the duration, or if using multiple nodes are preferred due to structural preferences, multiple auxiliary indexes may be used. Instead of routing all the incoming documents to a single index, they may use any partitioning function to route incoming documents to the auxiliary indexes.

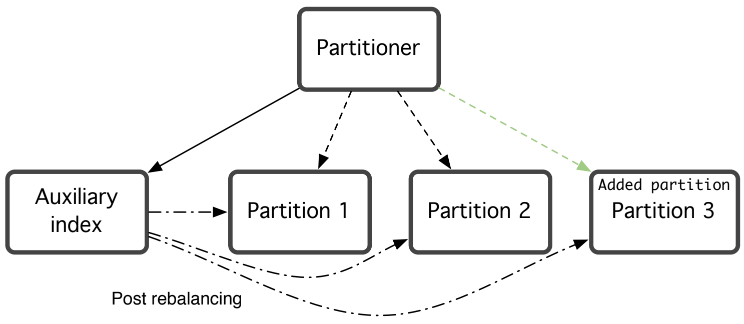


Figure 6.6: Freezing during repartitioning

An additional step that partitions the documents that have been put in the auxiliary index into their correct partition needs to be taken if there is no one-to-one mapping between the auxiliary indexes and the new partitions.

Replay-based repartitioning

Messages may be queued to disk and replayed at a later time if the indexing services are unavailable due to repartitioning (see Figure 6.7).

1. Route incoming messages to a serializer. The serializer may for example compress the messages an store them in a flat file.
2. When the indexing service is ready to accept messages again, the serialization can be reversed and the messages can be sent pushed to their intended destination.

Replaying should be performed by a layer in front of the partitioners to make sure the replayed messages use the updated partitioning services.

Replaying significantly increases the time between when a document is sent to the first processing node and the index queue and flush confirmation messages are published by the index writers. If the processes that feeds the system with documents rely on timeouts, it should to be informed that repartitioning is being performed and the timeout adjusted in order to avoid creating duplicate work, which would only create further contention and unnecessary processing of these duplicates.

6.3 Rebalancing

We define rebalancing as the act of moving documents to the correct partition as defined one or more the active partitioning functions.

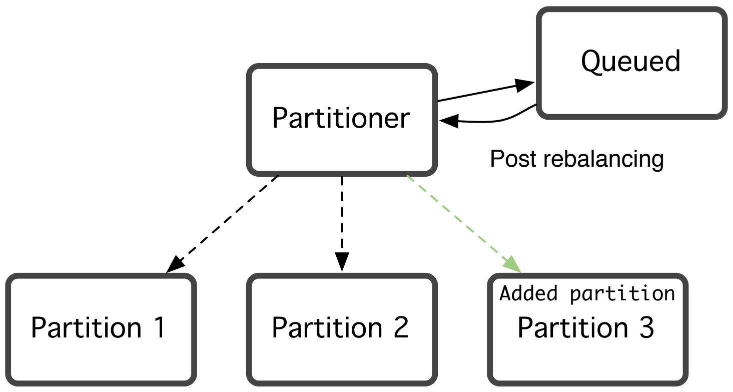


Figure 6.7: Replay-based repartitioning

Some partitioning functions are less suited for rebalancing. Hash-partitioning functions are not well suited for rebalancing, since increasing the modulus from n to $n+1$ results in each of the n previous partitions only containing $\frac{1}{n+1}$ correctly partitioned documents (see Figure 6.8).

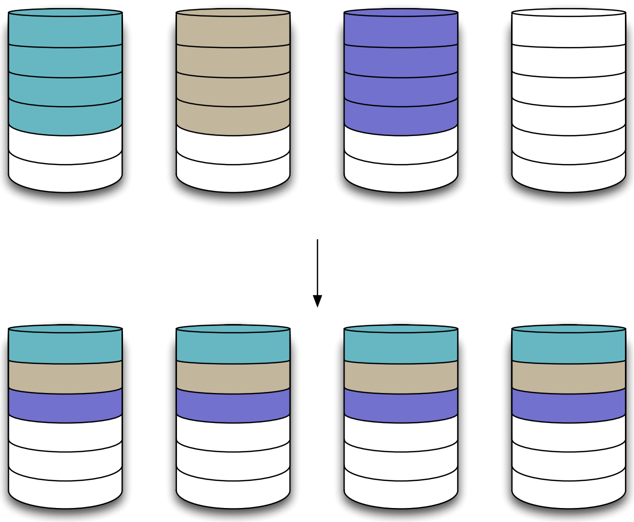


Figure 6.8: Rebalancing with a hash-partitioned function.

Depending on why we are repartitioning and what kind of partitioning functions are used, rebalancing can be either completely irrelevant or very important.

If the partitioning function used is non-repeatable, it is impossible to restrict queries to specific partitions. Thus, rebalancing will only be important if we wish to normalize the load between all the partitions. Dividing the load exactly equal between the partitions requires either a-priori knowledge of the query load or the partitions need to be rebalanced according to query analysis [50]. An approximation is to assume that every document is equally relevant to the query load as a whole and move an equal fraction of randomly selected documents from each existing partition to the new partitions.

Even if the partitioning function is repeatable, rebalancing does not necessarily have to be performed immediately after the partitioning function has been changed, or at all. In this scenario a strategy for handling consistency (see Section 6.4) should be implemented. Even though the system was repartitioned to increase the available write capacity or throughput, the read throughput may be fine and performing the rebalancing could potentially slow the system down. In these cases, the rebalancing may be deferred to a time where the system is under lower load and the rebalancing is unlikely to affect the users of the system

6.3.1 Prerequisites

In order to be able to effectively perform rebalancing between different partitions, each partition needs to know the previous operation it performed on a document, and when that operation was performed.

During indexing, each document is assumed to have a timestamp. This may either be set to the last-modified time by the source or it may be computed by one of the nodes in the network.

Having it set by the source is preferable as it is authoritative, but often a good enough approximation can be found by using the local clock of one of the nodes. If the timestamp is set by a singleton facade module, we can skip synchronization, but in a more fault-tolerant system there will be multiple facades and entry points, and the clocks would have to be kept in sync with each other by for example NTP [70], which can usually maintain a synchronized time within 10 ms over the Internet, and as low as 200 microseconds on a LAN. We store a mapping from the unique document identifier to some information that includes the latest operation performed and its timestamp to a b-tree of metadata in the full-text index.

Another solution is to use vector-clocks [71] or Lamport timestamps [72] as a way of preserving the order of updates and deletes.

This information may be stored as metadata in the postlist table using the unique term (see Section 5.6.1) as the key in order to be able to look up in the metadata for any given unique term in $\mathcal{O}(\log n)$ time.

6.4 Consistency

We start by defining a set of consistency anomalies that needs to be addressed.

Temporary loss means that a document has disappeared from any visible query results for an unspecified amount of time, but the document will reappear again at a later time without intervention.

Permanent loss means that a document that was acknowledged as indexed no longer exists in an index and will not reappear without manual re-indexing.

Temporary duplication means that a document exists in two or more indexes at the same time, and it could appear multiple times in the search results, depending on the degree of duplication, but the duplicates can be identified and removed the containing indexes.

Permanent duplication is the same as a temporary duplication, but there is no way of identifying which of the indexes has the duplicates.

The only way to recover from a permanent document loss is to re-index the document from the source.

It is possible to recover from permanent duplication by running a deduplication process, either as a standalone process or as part of a rebalancing process (see Section 6.4.1).

A central decision regarding consistency is deciding when a document gets removed from the source index. This decision, as we will see in the following sections, determines which of the above anomalies we may experience.

In the following section, the following shorthand notation is used:

i_commit - The index operation is committed.

d_commit - The delete operation is committed.

$\langle$**A, B**$\rangle$ denotes the time between the event A occurs until event B occurs.

6.4.1 Removing as part of the rebalancing

Documents may be removed from their source partitions as a part of the rebalancing process. We define the following deletion strategies:

Delete-early - deleting the document from the source index when the document is queued to either the partitioner or the destination index.

Delete-when-queued - deleting the document from the source index when the destination index has added the document.

Delete-after-flush - deleting the document from the source index when the destination index has flushed the document to the disk-based index.

The following three sections discuss these strategies in further detail.

Delete-early

In delete-early (Figure 6.9), document are temporarily lost from the they are sent from source index until the destination index commits the document. If the document is lost in-transit, the document may be permanently lost.

	Loss	Duplication
Temporary	⟨d_commit, i_commit⟩	⟨i_commit, d_commit⟩
Permanent	On index failure	–

Figure 6.9: Delete-early

Document duplication is possible using this technique, but duplicates will only exist until the source index commits.

Delete-when-queued

In delete-when-queued, documents are temporarily lost from they are queued by the destination index until the destination index commits the document.

Permanent document loss is possible if the destination index crashes after the document has been queued, but before it commits. Permanent document duplication is possible if the source never receives the queued-message from the destination.

	Loss	Duplication
Temporary	⟨d_commit, i_commit⟩	⟨i_commit, d_commit⟩
Permanent	On index failure	On queued-message loss

Figure 6.10: Delete-when-queued

Documents may be duplicated temporarily if the destination flushes the indexing operation before the source flushes the delete operation.

Temporary document losses may occur if the source flushes the delete operation before the destination flushes the indexing operation.

These two consistency anomalies are avoidable if both the source and the destination indexes commit in synchronization. Using a two-phase commit [6] it is possible avoid these, but this requires support in the underlying full-text index services which is non-existing at the time of this writing.

Delete-after-flush

In delete-after-flush (Figure 6.11), documents are duplicated from the destination commits the indexing operation until the source commits the delete operation. Permanent duplication is possible if the source crashes or rollbacks.

	Loss	Duplication
Temporary	–	⟨i_commit, d_commit⟩
Permanent	–	On deletion failure

Figure 6.11: Delete-after-flush

Handling replicas

Due to the asynchronous nature of the replication between the full-text indexes and the full-text replicas, it is possible that two full-text indexes that were flushed synchronously will be replicated in at an arbitrary time. This negates any effect gotten from using any kind of commit synchronization between the index nodes.

A solution to this issue is to either use some kind of two-phase commit in the index replication, or take use a delete-after-sync strategy that only deletes after all the required replicas have completed one round of synchronization after the previous flush.

6.4.2 Removing as an integral part of indexing

The deletion-handling can be addressed by the standard indexing process if the same nodes and pipelines are used by the rebalancing process in order to move the documents.

A potential issue with these technique is that if a partition is offline when messages informs the other partitions about performed operations are published, it will not know that another partition has indexed one of its documents and will not delete it. It is possible to minimize the chance of this happening by configuring the high water mark and considering swapping to disk in the publishing queue so that the partitions will receive messages that has been queued when it was offline. However, due to the nature of the problem, there is no guarantee when the partition will come online again and no guarantee that no messages have been lost just as the disconnect happened. Thus, regular maintenance should be performed (see Section 6.4.4).

If this technique is used together with a rebalancing process that relays rebalanced documents to the partitioner it may suffer from temporary document disappearance if the delete operations are processed quicker than the indexing operations. Additionally, if indexing fails on the document for some reason, it is permanently lost and will have to be reindexed.

The delete operation can be generated by either the partitioner or the indexes as described in the next two sections.

	Loss	**Duplication**
Temporary	$\langle$d_commit, i_commit$\rangle$	$\langle$i_commit, d_commit$\rangle$
Permanent	On index failure.	On deletion failure

Figure 6.12: Removing as part of the indexing by broadcasting.

Partitioner broadcasting

When a partition is chosen for a document, the partitioner can send a delete message to all the other partitions attempting to ensure that the document will only exist in at most one index at a time.

Documents are temporarily lost if the delete operation is flushed before the index operation is flushed and permanently lost if the index operation fails.

Node broadcasting

When a partition flushes a document, it can publish message containing information about the operation performed, the unique identifier of the document and the timestamp of the document. All partitions can subscribe to these messages and act accordingly: if another partition reports flushing the indexing of a document, we can delete that document from our partition.

In order to subscribe on these messages, the partitions must then either have knowledge of each other or a forwarder may be used.

6.4.3 Removing post-rebalancing

Documents may be removed after all the rebalancing has taken place. This causes a temporary spike in hard disk space usage as both the old documents and the new documents are stored in the indexes at the same time and slightly longer rebalancing time as the B-trees gets bigger than in the previous techniques.

A benefit of only removing documents after the rebalancing is completed is higher throughput during the rebalancing. This is because fewer operations are performed and no synchronization or communication between the indexing nodes are required. If the rebalancing fails or stops for whatever reason, it may be restarted from the beginning or cancelled without any documents being in any danger of getting lost.

Massive document duplication is to be expected this way, as every document that gets rebalanced will yield one duplicate during the rebalancing process as a result of the source index sending a copy to the destination index.. These duplicates will be safely removed as the post-rebalancing removal process completes.

6.4.4 Partition maintenance

Partition maintenance is about making sure the partitions are healthy and only include the documents it is supposed to contain. The partition maintenance process can be made sure to only run when all the partitions are available by invoking it manually when the partitions are known to be up, and this section assumes that this is the case.

Depending on the consistency requirements and the methodologies used from Section 6.4, documents that are discovered to be in the wrong partition can either be the result of an indexing request that have not been successfully been committed a lost queued or flushed document message. By looking at the metadata from the source and destination partitions it is possible to determine whether the situation was a result of a failed reindex or simply the loss of a queued or flushed document message.

If the correct partition contains the document or deletion metadata that is the same or fresher than the source partition, it may safely be deleted. Otherwise, it should be classified according to the consistency methodology and handled accordingly. For example, it may be classified as a lost document indexing message and either attempted repartitioned immediately or marked for repartitioning at a later time.

6.4.5 Query-time de-duplication

Some of the techniques outlined in the previous sections results in a document being indexed in two partitions at the same time, which results in duplicates being returned by queries. In order to compensate for this when using xapian remote databases, we may store the unique term in a value field in order to be able to be able to filter out duplicates.

Each of the partitions may be queried separately and manually merge the results from each of them as an extra step in the query processing pipeline. This is the only way to remove duplicates when using broker-based querying (see Sections 5.8.2 and 5.8.3) and is possible to use independently of the type of full-text index used.

If querying uses direct-access (defined in Section 5.8.1), we can *collapse* over the unique term in order to filter the duplicates.. Collapsing is a feature in Xapian that enable us to limit the number of documents returned with each particular value of the field that is collapsed on. Collapsing over the unique term field with a max limit of 1 per unique value will effectively remove any duplicates from the matching sets returned from Xapian. However, it is not currently possible to collapse over multiple fields in Xapian and if collapsing is used to implement other features such as category searches, manually merging is mandated.

6.4.6 Important deletes

During online rebalancing, a delete operation may be sent to the new index writer before the previous index writer has deleted its document. In many cases this may be written off as a temporary side effect of the rebalancing, but if it is important that the delete is handled quickly, extra care has to be taken.

A solution to this is to let the partitioner always broadcast delete messages. Delete messages are small and only includes the verb and a unique identifier, which does not add much to the network traffic. For the delete to trickle down to the replicas, the index writer has to commit and flush. This technique is not usable with pseudo-random partitioning using load balancing without adding additional queues as there is no way to perform a load-balanced broadcast.

Another solution that may be used in conjunction to the former is to let query nodes subscribe to delete messages and filter out deleted documents before returning them (see Figure 6.13). This complicates the query nodes slightly however, and adds some overhead on the most performance-critical part of most search system and should be implemented with due consideration for execution time.

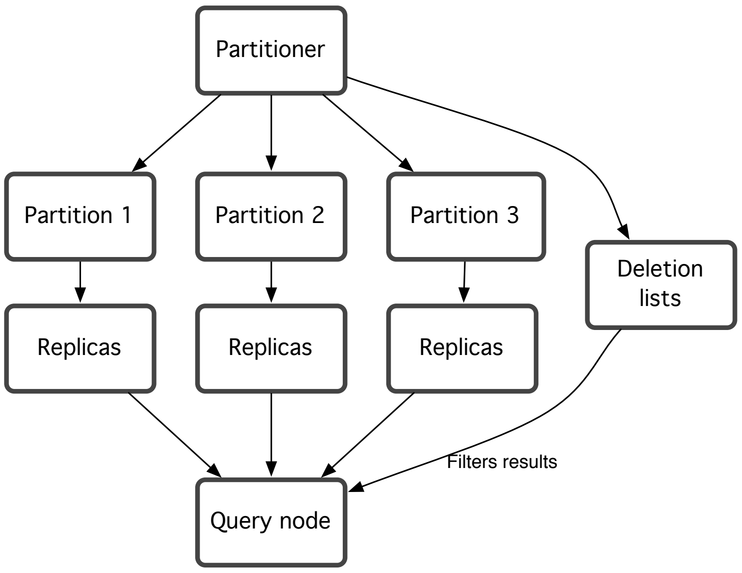


Figure 6.13: Using delete lists to filter query results.

An artifact of implementing this is that the exact page count and result count may be unknown to the query nodes until they have been counted and filtered. Since the full results are never returned and processed by the query nodes, this adds an additional error-source when estimating the number of hits for any given query. However, it does provide explicit control over how the merge is performed and which result object is returned

Chapter 7

Implementation

In this chapter, we will define the components of a *node*. An overview of the implementation is shown in Figure 7.1.

A node is a collection of *pipelines* that uses *processors* to process input. *Resource providers* provide both *processors* with the resources they require to operate and the pipelines with data to be processed.

The implementation is heavily inspired by flow based programming (see Section 3.3).

Full source code for this implementation is available from the author upon request, and a public developer preview is expected during Q2 2011.

7.1 Processors

A processor is a configurable component that operates on a *baton* (described in detail in Section 7.2). It takes some initial parameters and is later configured with a *runtime environment* (see Section 7.5) before it is used to process batons.

```
class IPipelineProcessor(interface.Interface):
    name = interface.Attribute('Short name, used in configuration files.')

    def configure(self, runtime_environment):
        """ Configures the processor with the runtime environment.

        This is where the processors usually will request dependencies and prepare
        itself for processing batons.
        """

    def process(self, baton):
        """ Processes a baton.

        This function is called by the evaluator when it is time for a processor to
        process a baton. The return value of this function is used as the input baton
        of consumers of this processor.
        """
```

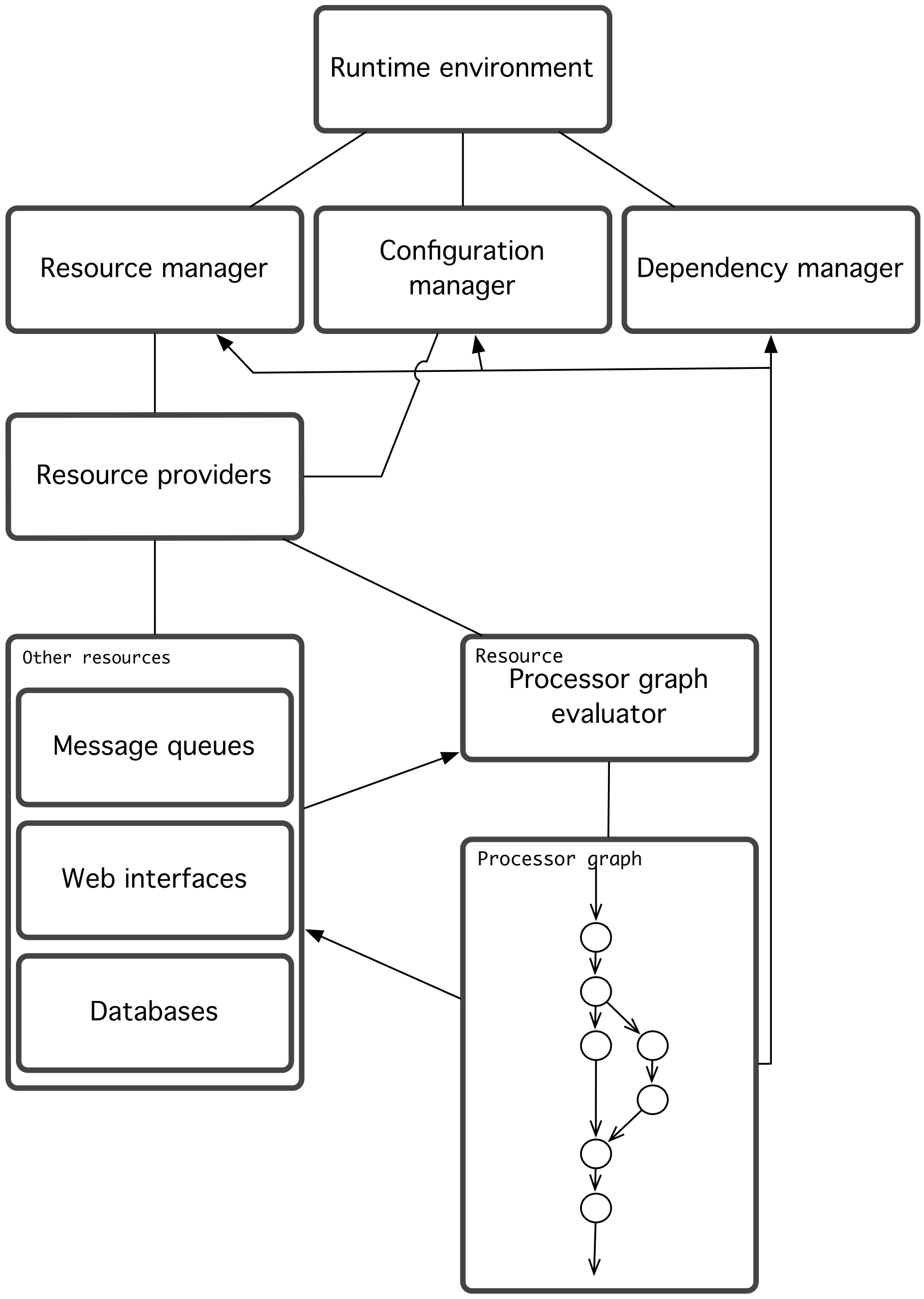


Figure 7.1: Implementation overview.

7.2 Batons

A baton is a unit that is passed through the *processing graph* to the *processors*, and is the equivalent of an Information Packet in flow-based programming. The type and content of a baton is entirely defined by its source and the pipeline it passes through, as each processor may perform arbitrary operations on it, or even return a completely new baton that replaces the baton for any downstream processors in the processing graph.

The baton is most commonly a python dictionary instance.

```
{
    'id': 1234567890,
    'updated': datetime.datetime(2011, 3, 13, 15, 18, 15, 923888),
    'title': 'Example title',
    'text': 'This is the contents of this document.'

    # the following keys are examples that may be computed by processors
    # in the pipeline based on the above keys:
    'unique_term': 'ID1234567890',
    'terms': {
        'title': [IndexTerm(...), IndexTerm(...)],
        'text': [IndexTerm(...), ...]
    },
    'document': xapian.Document(...),
}
```

After a text-processing node has processed a document, extraneous keys are removed and the baton is prepared for serialization (see Section 3.5.2). Since most serialization libraries do not support serializing arbitrary objects, Xapian documents must be serialized by using

`xapian.Document.serialise()`

before the baton itself is serialized.

Below is an example baton after the serialization, using JSON:

```
{
    "document": "\\u0000\\u0002\\u0007example\\u0001\\u0000\\u0005title\\u0001...",
    "unique_term": "foo"
}
```

The serialized baton is then ready to be sent to an indexing node, which will unserialize the baton and the Xapian document as the first processing steps.

7.3 Processing graph

A *processing graph* is a graph of connected *processors*, each of which may have consumers and error-consumers (see Figure 7.2 and 7.3). Consumers are other processors that will receive and process returned data and error-consumers are consumers that are used if the processor raises an exception. This loosely resembles the well-known “Deferred” pattern [73] used in Twisted and many frameworks, both commercial [74] and open source [75].

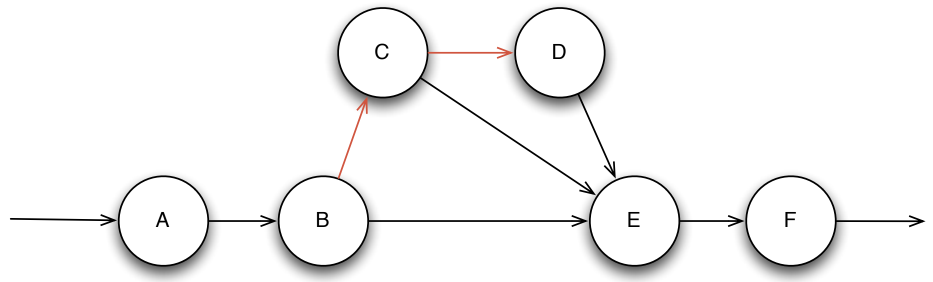


Figure 7.2: A pipeline. Some processors have error consumers, marked with red edges.

```
A()
try:
    B()
except:
    try:
        C()
    except:
        D()
E()
F()
```

Figure 7.3: Pseudo-code equivalent of the processing graph shown in Figure 7.2

7.4 Processing graph evaluator

An evaluator is used to move the baton through the processing graph. A processing graph essentially declares *what* the processing does while the evaluator defines *how* the processing is done.

Using different evaluators makes it possible to only store debugging or profiling information when required, thus reducing execution time for the common case.

An evaluator that operates on a processing graph is called a *pipeline*.

Different evaluators makes it possible to define different kinds of pipelines, for example asynchronous, coroutine-based [76] and/or stateful pipelines. It is important to note that even while the pipeline evaluator does not carry any state by itself, it is possible to write stateful processors by wrapping coroutines or using variables that persist between invocations of `process(baton)`.

7.5 Runtime environment

The *runtime environment* is used by processors to access some central components of a running system, such as the *configuration manager*, the *dependency manager* and the *resource manager*.

7.5.1 Configuration manager

The configuration manager is responsible for loading and providing access to the configuration. The configuration files are written in YAML [77] that contains mappings that translates into python `dict` objects. The configuration dictionary is used to configure resource providers, pipelines and processors.

7.5.2 Resource manager

A resource manager is used to load *resource providers* as plugins. Resource providers register the *resources* they can provide with the resource manager, which is later used by other resource providers or processors.

Resources may be live objects such as database connections, objects providing functionality such as decompounders [78], or static resources such as dictionaries.

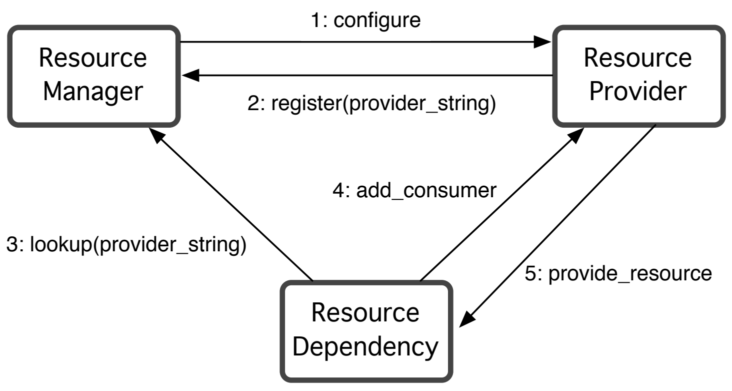


Figure 7.4: The resource manager configures resource providers which provide the resource dependencies with resources.

The most important resources used by the indexing nodes in the benchmarking (Chapter 8) are:

Message queues (see Section 3.5) which are used to publish information about indexed and deleted documents.

Xapian indexes that are used to index and store the documents.

Shared contexts which are in-memory dictionaries that are used to store the unique terms of documents that have been added to the index, but not yet been committed. This is essential in order to be able to inform other indexing nodes that documents have been flushed to disk, which is used by many of the techniques described in Section 6.4. The shared contexts are also used to store statistics which are used to generate performance graphs.

7.5.3 Dependency manager

Dependencies are registered with a dependency manager. A dependency may be an *instance dependency* (depending on a concrete python object instance), or a *resource dependency*.

Instance dependencies are immediately available as soon as all their dependencies are available. One example of this is that a pipeline is ready as soon as all its processors are.

Resource dependencies are dependencies that are identified by a provider string that is registered in the resource manager. The dependency manager uses the resource manager to resolve these dependencies. In contrast to the instance dependency, the resource encapsulated by a resource dependency may change without the dependency itself changing.

The dependency manager is responsible for maintaining a directed, acyclic graph of dependencies. If a dependency becomes unavailable, the event is cascaded up the dependency graph, giving its dependents a chance to decide if they can continue operating or not.

For example, if a database server goes down, all processors that depend on having a connection to that database becomes unavailable and the pipelines that use these processors becomes unavailable. The users of these pipelines, such as resource providers that provide batons from external sources can stop producing batons and wait for the resource to become available again.

This cascading property is central to use the flow control (see Section 7.7.4) available in ZeroMQ: a socket that is not being read from will eventually reach its configured high water mark that enables upstream nodes to either stop sending or redirect messages to other downstream nodes with available capacity.

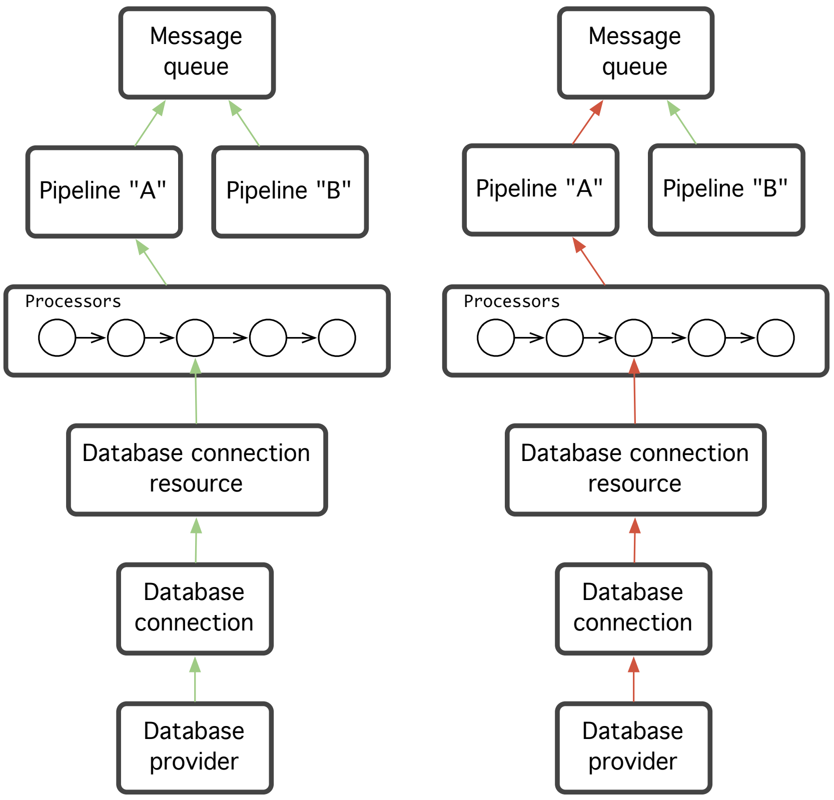


Figure 7.5: An example dependency graph. On the right side, the database connection has been temporarily lost, which cascades up to the message queues that use an affected pipeline.

7.6 Network of nodes

If all the state required to process a baton is stored in the baton itself, pipelines may be distributed by splitting the configuration at any processor.

The following is an example of distributing a single factorizer. In the single-node example, the factorizing is performed by the server and in the distributed example the factorizing is performed by a pool of transient worker nodes.

A simple factorizer can be configured like this:

```
# filename: simple-factorizer.yaml

pipelines:
    factorize:          # the input baton is an integer
        - find-factors # returns a list of factors
        - print-result
```

A distributed version of the above is in two parts. The first is the server configuration:

```
# filename: factorizer-server.yaml

zmq:
    queues:
        to_worker:      # used to push data to the workers
            type: PUSH
            binds: ['tcp://0.0.0.0:5000']
        from_worker:    # used to pull data from the workers
            type: PULL
            binds: ['tcp://0.0.0.0:5001']
            pipeline: from_worker

pipelines:
    factorize:
        - encode-json   # the message must to be a string, so we encode it with json
        - send-message: # sends the integer to a worker
            queue: to_worker
    from_worker:        # prints the result gotten from the worker
        - decode-json   # the worker return json-encoded results
        - print-result
```

The worker configuration:

```
# filename: factorizer-worker.yaml
zmq:
    queues:
        from_server:    # used to pull data from the server
            type: PULL
            connects: ['tcp://0.0.0.0:5000']
            pipeline: from_server
        to_server:      # used to push data to the server
            type: PUSH
            connects: ['tcp://0.0.0.0:5001']

pipelines:
    from_server:
        - decode-json   # the message is in json
        - find-factors
        - encode-json   # the message must be a string, so we encode it with json
        - send-message:
            queue: to_server
```

To run both these configurations in a single process, both configurations can be included in the same file:

```
# filename: factorizer-both.yaml

includes:
    - factorizer-server.yaml
    - factorizer-worker.yaml
```

This allows for running a distributed system in a single process, which greatly simplifies testing and debugging. One caveat is that it still is running as a single process, and if blocking operations in one pipeline may significantly affect the performance of the other pipelines.

7.7 Restructuring networks

Changes in the data volume or the way we want it indexed reveals the need to restructure networks (as defined in Section 3.6). This needs to be supported by building blocks, and is part of the reason why the processing graph is a good idea, as it makes for easy changes.

As seen in Chapter 3, the pipelines in a node and how the nodes are connected in a network determines how documents are indexed. To change the pipelines in a node or a set of nodes, we update its configuration file(s) and restart it. The affected node is unavailable during this process. The following sections discuss ways of gracefully performing these changes.

ZeroMQ does not as of this writing support graceful disconnects, which means some extra care has to be taken when the network has to be updated. A queue that gets closed cannot be read from further, which means messages that were in-transit when the close occurred are potentially lost. A sender can be shut down pseudo-gracefully by setting a high water mark to 1 and perform a blocking send. At that point, there is only one message that has not been sent. That message can then be tracked using the MessageTracker support in ZeroMQ, which presents an interface to receive notifications when a message has been successfully sent. When there are no unsent messages for a queue, it can be closed safely. New output queues can be configured and started simultaneously in order to avoid downtime in the node.

7.7.1 Queue swapping

It is possible to atomically update the configuration of multiple nodes by constructing a new network of nodes with the new configuration, and then change the output queues of the parent nodes in the network when the newly constructed network is ready. The new network may be configured to run on the same physical nodes as the previous network without interfering.

Parts of the previous network may also be reused when constructing the new network. Index writers are of particular interest to reuse since they require an exclusive write-lock on the indexes they operate on. Note that if nodes are reused, they will continue to use their old output queues, and thus may reduce the new networks applicability to testing.

Not all network changes are considered restructuring. For example, adding transient nodes such as filter and text processing nodes require nothing more than starting a process and connecting to the correct message queues.

7.7.2 Explicit pulling

ZeroMQ pull sockets are used as the receiving end of the pipeline processing pattern that provide automatic load balancing of messages when multiple downstream nodes are connected. Shutting down a pull socket is a two part problem. The first problem is that there is no easy way of telling whether there are more messages underway without hitting the high water mark, and even then there may be more messages in-transit due to latency.

The second problem is that even if it was possible to tell that there were no messages mid-transit, the lowest high water mark that is possible to set in ZeroMQ is 1. This means that there might be one message left in the PUSH sockets local output queue for the receiving PULL socket.

Instead of relying on the push socket to send messages downstream, it is possible to use an XREQ in the downstream node to explicitly pull messages from an upstream XREP socket. In order to be more effective, a small protocol for this to allow for multiple messages to be pulled at the same time could be defined. Furthermore, this could be built on top of a heartbeat service that could provide administrators with a better overview over the network as a whole.

7.7.3 Resending

A different approach is to publish messages at some or even all the nodes that contain information about the data it processes. For example, an indexing node can publish one message per document it adds to its index, and one message per document it has flushed to its index. A client feeds the system with data can subscribe to these messages and use the received messages to detect lost messages.

Depending on which nodes the client is subscribing on messages from it is possible to detect slow nodes because messages from the node are published at a significantly lower rate than its sibling nodes, crashed, hung or restarting nodes, which will not publish any messages at all. If one of the latter are detected and documents from the client are not seeing progress in the network, the client may assume the messages have been lost, either due to a crash, an unclean disconnect or due to a hung node. In either case, it is safe for the client to assume the worst and resend the affected documents.

This technique relies on the client having knowledge of how the network is configured, which may not be desirable. It is easy to avoid this by using client brokers, which can be used as facades to the system. The client brokers can be configured with a list of stages the documents can go through, where each stage represents a set of nodes in the network the document may be in.

When the client broker receives a document that should be indexed, it adds the document and the current time to the initial stage and sends it to the first node in the network. Processing messages that are published by the nodes in the network are used to update the document state. If a document is stuck in a single stage for an unproportionally long time, the document may be either automatically resent if the client broker still has a copy of the document, or the client may be informed.

It is important to always look at the current average processing times on a node, as we do not want long processing times on several processors due to high load to trigger resending.

In order to tell the difference between a failed node and a node that is waiting for available capacity in downstream nodes, the nodes must periodically publish "pending" messages when documents are busy waiting for an available downstream node which the client broker can use to update the timestamp of the document in its current stage. Otherwise, if all the nodes in the following stage were busy or stopped, all documents would be marked as failed and possibly resent, which would only add duplicates to the backlog of documents, as these pending documents would automatically continue their processing once the downstream nodes has some available capacity.

7.7.4 Flow control

In Section 3.6 we defined networks of nodes connected by some kind of messaging layer. Even though the messaging layer could be asynchronous, the requirements for flow control should be carefully considered. If the upstream nodes are allowed to process data as quickly as they can without regard to the throughput of downstream nodes, memory usage and performance of the downstream nodes may be difficult or even impossible to control.

Even when using the high water mark support in ZeroMQ, care must be taken to only read messages from the sockets when sufficient memory and other resources are available to perform the operations the relevant pipeline defines.

Chapter 8

Benchmarks

This chapter provides benchmarks of the implementation described in Chapter 7 using the different strategies suggested in Chapter 6. All the benchmarks were performed using Amazon EC2. The instance type used is described in each section and an overview of the instance type specs are found in Figure 8.1.

Name	**API name**	**Memory**	**Cores**	**ECUs/core**
Extra Large Instance	m1.xlarge	15 GB	4	2
High-Memory Extra Large Instance	m2.xlarge	17.1 GB	2	3.25
High-Memory Quadruple XL Instance	m2.4xlarge	68.4 GB	8	3.25

Figure 8.1: EC2 instance types used during benchmarking.

The dataset is described in Section 8.1 and the initial indexing is described in Section 8.2.

8.1 Dataset description

The dataset used for benchmarking comes from the Freebase Wikipedia Extraction (WEX [79]), which is a processed dump of the English language Wikipedia. The WEX dataset contains the markup for each article, transformed into machine-readable XML. Some elements, such as templates, infoboxes, categories, article sections, and redirects are extracted in tabular form.

The XML, even though it is machine-readable, does not follow an exact schema because there is no standardization on Wikipedia itself on how the mediawiki markup should be used to format for example dates, monetary units and so on. Because of this, only the title, text and updated timestamp were used, as these were the only attributes found to exist in all of the articles.

The message transport was benchmarked by using to small instances within the same availability zone. A small document collection of 73227 documents were transported in 41 seconds, resulting in an average of $73227/41 = 1786$ documents per second. Experimenting with the high water mark showed no significant difference between test runs with a high water mark of 10, 100 or 1000, as each run were within 2 seconds of each other. With a collection size of $855MB$, this gives a throughput of just above $20MB/s$ between the nodes. Any indexing throughput higher than this are thus unlikely to be achievable with only one data source. Using the internal or external IP addresses of the nodes made no difference. The remaining benchmarks were run with a high water mark of 100.

8.2 Indexing

An initial index was built using a single EC2 Extra Large instance (m1.xlarge). Load-balanced partitioning 5.4.5 were used to create 8 partitions of approximately equal size (see Appendix A for the configurations used during indexing).

Figure 8.2 shows the throughput and uncommitted document count of a single partition and the combined I/O throughput of the indexing system. The CPU usage (not shown) were 100% during the whole indexing process.

8.3 Offline repartitioning

Using an Amazon EC2 High Memory Quadruple Extra Large instance (m2.4xlarge), the 8 indexed partitions were mounted and split in half by 8 parallel processes executing the following code:

```python
for i, unique_term in enumerate(from_index.metadata_keys()):
    # construct the document
    document = document_from_unique_term(unique_term)

    # add it to the destination and remove it from the source
    to_index.add_document(document)
    from_index.delete_document(unique_term)

    # only process half the documents:
    if (total_docs/2) <= i:
        break

    # commit when halfway done ..
    if i == total_docs/4:
        yield commit_databases_in_parallel(from_index, to_index)

# .. and commit when done, resulting in a total of two commits.
yield commit_databases_in_parallel(from_index, to_index)
```

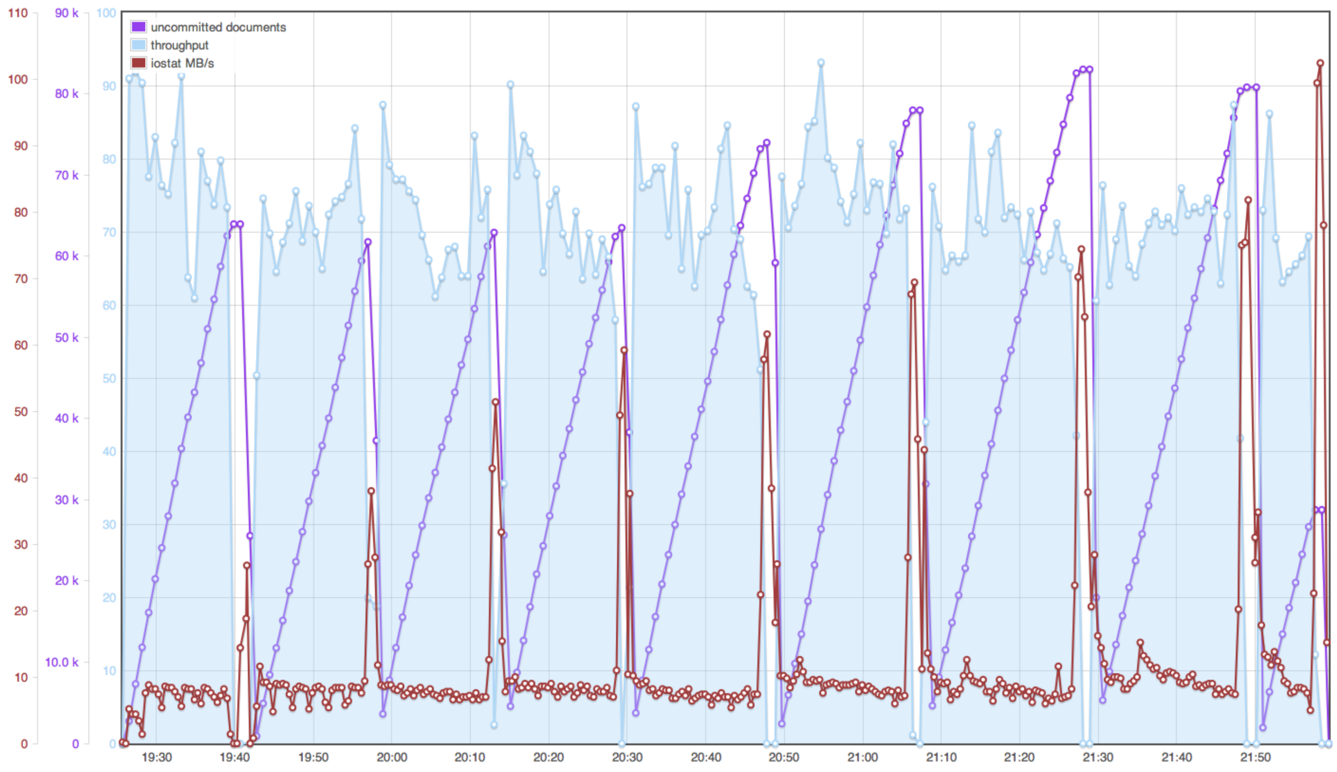


Figure 8.2: Indexing performance on m1.xlarge instance. The throughput and uncommitted document counts are per node. A total of 8 nodes were used on a single machine.

Figure 8.3 shows the time taken to perform the offline repartitioning. The formatting and mounting time of the added partitions is omitted as this may be done before the existing partitions are taken offline.

Action	**Time**
Attaching the required EBS volumes	2m7.29s
Splitting 8 partitions into 16	26m34.18s
Unmounting the volumes	18.08s
Detaching the volumes	3m38.11s
Total	**32m37.66s**

Figure 8.3: Time taken to rebalance 8 partitions into 16 partitions using offline repartitioning.

8.3.1 Merging

The 16 partitions created in the previous section was merged with 8 concurrent processes using “xapian-compact”, a tool provided by Xapian that compacts or merges one or more databases.

Below is the output from one of these partitions:

```
$ xapian-compact wex01.xapian wex09.xapian wex_compacted.xapian
postlist: Reduced by 31% 232600K (742336K -> 509736K)
record: Reduced by 1% 5816K (395728K -> 389912K)
termlist: Reduced by 14% 83896K (560720K -> 476824K)
position: INCREASED by 0% 1576K (2503544K -> 2505120K)
spelling: doesnt exist
synonym: doesnt exist
```

The total time taken by xapian-compact to merge the databases were 8m22.48s. Assuming the same overhead on the attaching, detaching and unmounting as in Figure 8.4, merging 16 partitions into 8 partition gives:

Action	**Time**
Attaching the required EBS volumes	2m7.29s
Merging 16 partitions into 8	**8m22.48s**
Unmounting the volumes	18.08s
Detaching the volumes	3m38.11s
Total	**14m15.96s**

Figure 8.4: Time taken to merge 16 partitions into 8 partitions using offline repartitioning with xapian-compact.

The big difference in the time taken to split and merge these partitions can be attributed to the fact that xapian-compact operates on the internal Xapian database structure and can perform optimizations that are not otherwise available to users of the standard Xapian API.

8.4 Online repartitioning

The following sections benchmarks the online strategies proposed in Chapter 6. The basis for these operations is an indexed version of the WEX document collection. One partition, approximately $\frac{1}{8}$ of the WEX document collection, which is equal to $600k$ documents were split in half in order to simulate going from 8 to 16 partitions. In a real-world system, the eight partitions would be rebalanced in parallel, independent of each-other, resulting in a total rebalancing time approximately equal to the rebalancing time of a single partition.

The source and destination nodes were EC2 High-Memory Extra Large Instances (m2.xlarge). Each processing node had 3.25 ECUs available, which is higher than what was used in the offline repartitioning benchmark that used a total of 8 virtual cores with 3.25 ECUs each. This reflects the fact that a distributed system has more processing power available as distributed resources, and should be taken into consideration when comparing the results from the offline and online benchmarks.

During benchmarking, no messages were lost and no node failures were experienced.

The overhead it takes to create the required new instances and volumes is not considered, as it may be performed before the online repartitioning and rebalancing process is started.

All the rebalancing operations were performed without the need of intermediary commits and the commit was performed after all the required operations were performed on the current index.

The throughput in most of the figures found in this section has a camel-hump shape that is the result of two batches of documents of shorter size, such as category pages and short description pages which are naturally occurring in the document collection.

When the benchmark required a publish-subscribe message queue (see Section 3.5.1), no high water mark were used as we do not want to block the rest of the processing waiting for these messages to transfer. In a large real world system, these are likely to be offloaded to a dedicated forwarding node that swaps messages to disk after reaching an internal high water mark for each subscriber.

The complete index configurations and rebalancing process can be found in Appendix B.

8.4.1 Delete early

Using delete-early, the documents were deleted from the source index immediately after being sent to the destination node. No further communication were required between the nodes. The source was flushed immediately sending the last rebalanced document to the destination node. The total rebalancing time using this strategy was 23m25.53s (see Figure 8.5).

Action	**Time**
Splitting the source index	21m46.97s
Committing the indexes	1m38.56s
Total	**23m25.53s**

Figure 8.5: Time taken to split 8 partitions into 16 using delete-early.

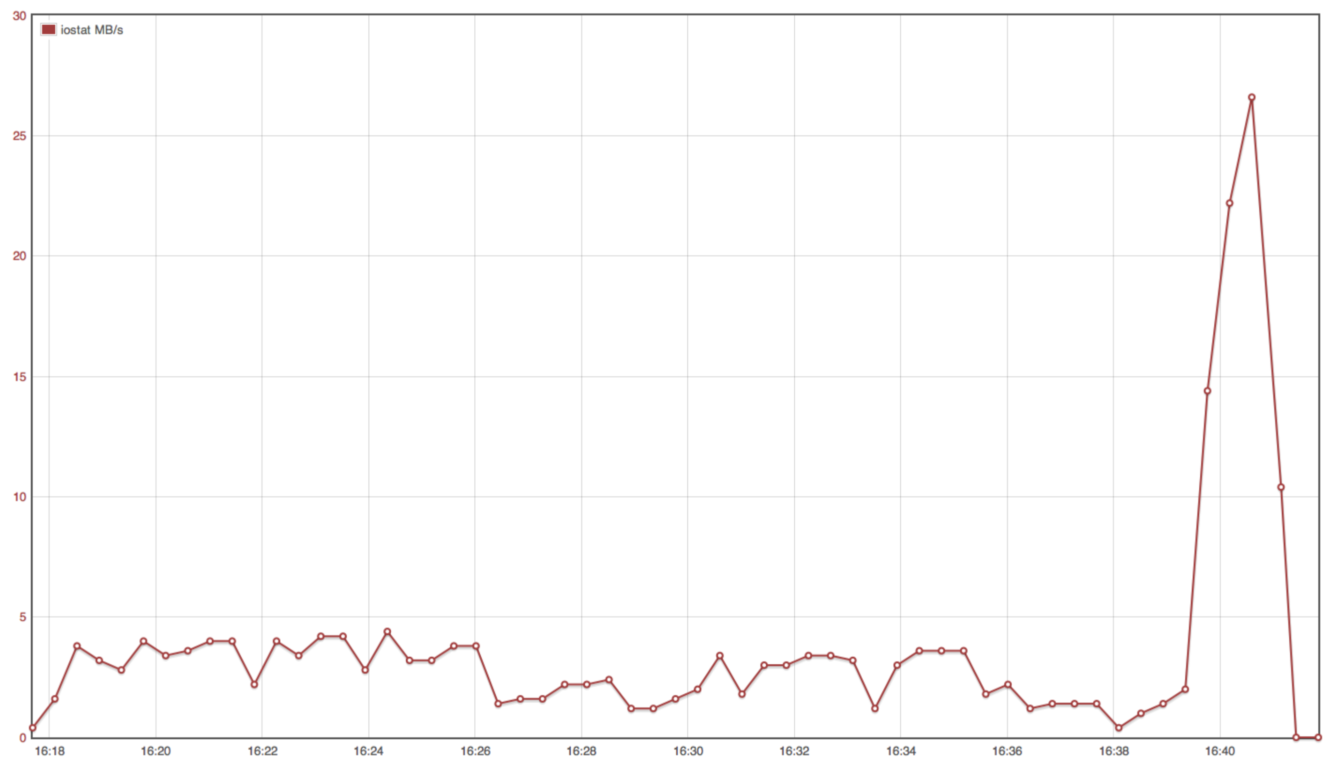


Figure 8.6: Delete early: source.

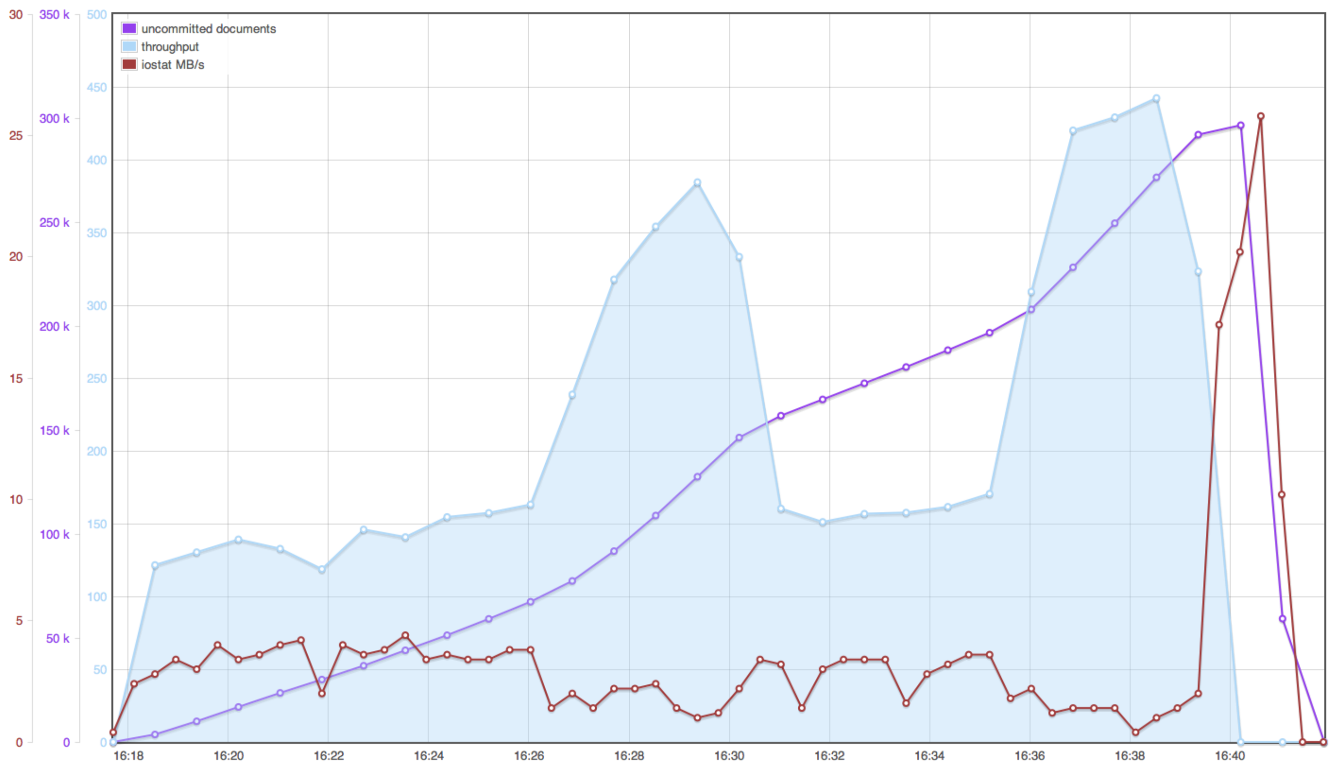


Figure 8.7: Delete early: destination.

This is faster than the offline partitioning was able to complete the task, and the result of having more processing power available in a distributed system than on a single server. Figure 8.6 and 8.7 shows the performance of the source and destination nodes respectively.

8.4.2 Delete when queued

Using delete-when-queued, the destination node was configured to publish a message containing the unique term for every document it queued to its index. These messages were subscribed on by the source, and the relevant document were queued for removal as the published messages arrived. The total rebalancing time using this strategy was 23m17.98s (see Figure 8.8).

Action	**Time**
Splitting the source index	18m56.56s
Waiting for “queued” messages	2m35.16s
Committing the indexes	1m46.26s
Total	**23m17.98s**

Figure 8.8: Time taken to split 8 partitions into 16 using delete-when-queued.

Figure 8.9 and 8.10 shows the performance of the source and destination nodes respectively using this strategy.

8.4.3 Delete after flush

Using delete-after-flush, the destination node published a message after every flush containing the unique term for every document that was flushed. As in the previous section, these messages resulted in the source queueing the removal of the documents as they arrived. The total rebalancing time using this strategy was 23m17.98s (see Figure 8.11).

Action	**Time**
Splitting the source index	18m56.56s
Committing the destination index	1m21.00s
Waiting for “commit” messages	10m10.92s
Committing the destination index	1m5.25s
Total	**31m36.11s**

Figure 8.11: Time taken to split 8 partitions into 16 using delete-after-flush.

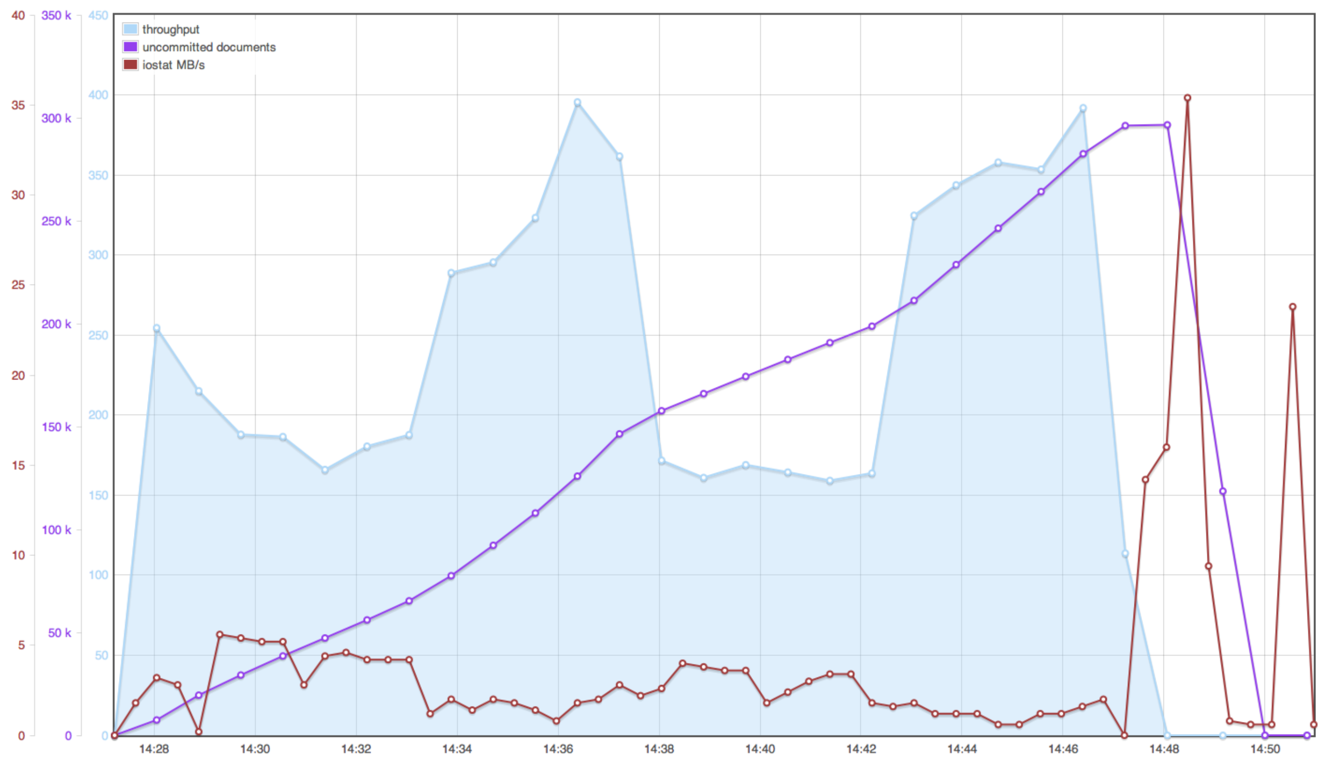


Figure 8.9: Delete when queued: source.

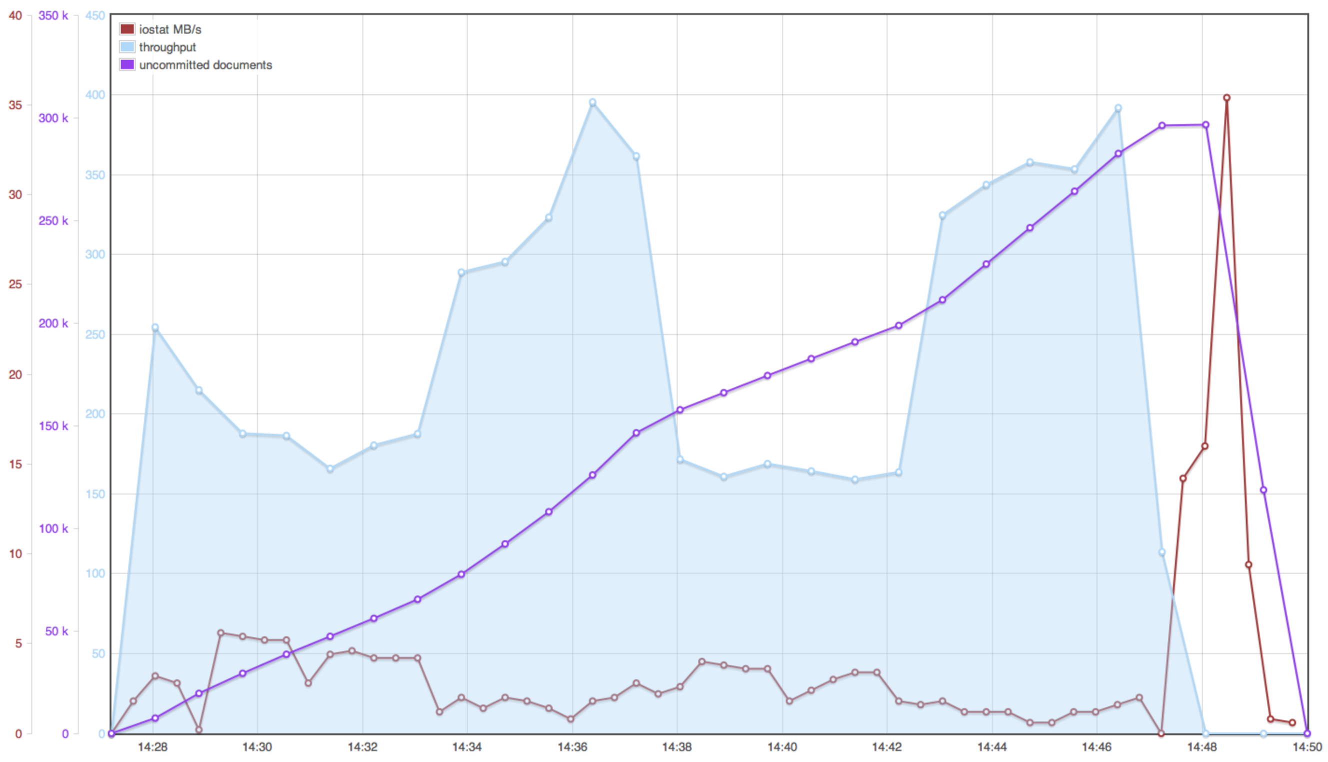


Figure 8.10: Delete when queued: destination.

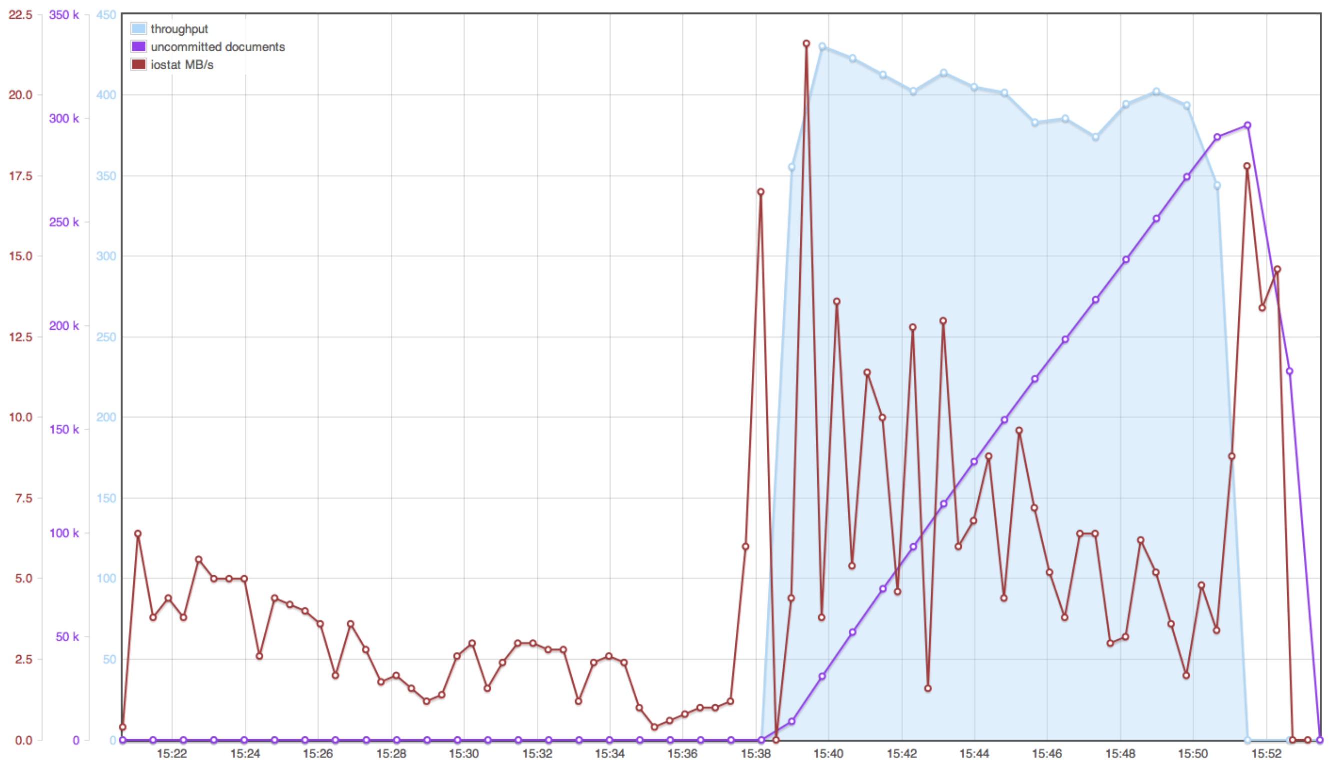


Figure 8.12: Delete after flushing: source.

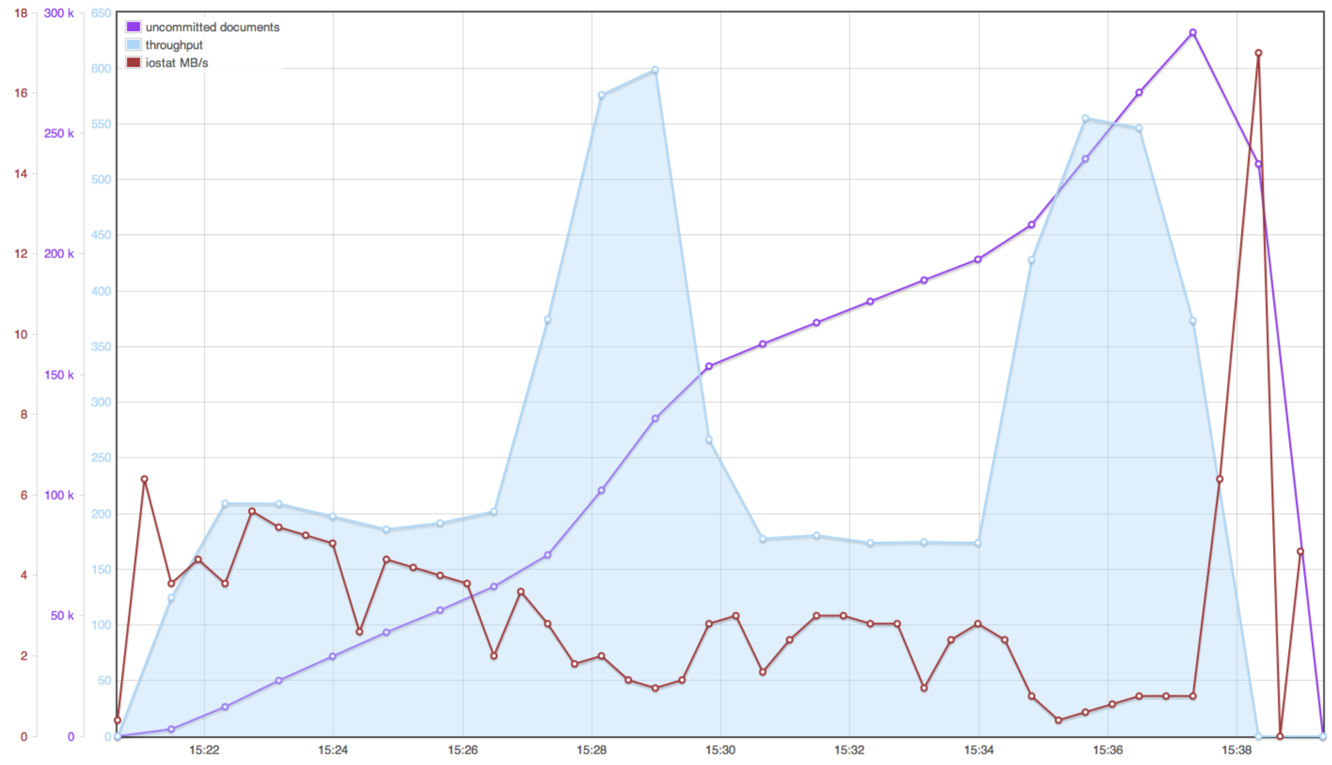


Figure 8.13: Delete after flushing: destination.

Using this strategy, the source partition had to defer any processing of document deletions until the destination had committed. This removed the possibility that the delete operation and sending of documents could be done in parallel, resulting in a higher total rebalancing time than the previous two strategies. Figure 8.12 and 8.13 shows the performance of the source and destination nodes respectively.

8.4.4 Merging

Merging two indexes into one was performed by telling one of the nodes to send all its documents to the other node. In contrast to the splitting that was performed in the previous sections, merging does not require any deletes. When the merging process is complete, the source volume is simply discarded.

Action	**Time**
Merging the indexes	26m34.82s
Committing the destination index	1m53.32s
Total	**28m28.14s**

Figure 8.14: Time taken to merge 16 partition into 8 using online repartitioning.

As seen in Figure 8.14, the time used by the merging and committing is slightly higher than the splitting time in the previous sections. This is a result of a non-uniform collection, as short documents were not as frequent in the remaining half of the first partition. The commit-time is also a little larger as a result of committing in a larger document collection, which requires more disk-seeks to complete.

8.5 Comparison

Figure 8.16 shows the total processing time of the benchmarks in the previous sections. Online repartitioning using *delete-when-queued* was the quickest, closely followed by *delete-early*. Offline and delete-after-flush used approximately the same amount of time.

A real deployment would have to carefully consider the consistency demands outlined in Section 6.4 before selecting a strategy.

An argument for using offline repartitioning is that it requires no changes to the indexing nodes and has no overhead during indexing.

Delete-when-queued and delete-after-flush can both handle removal as an integral part of indexing, described in detail in Section 6.4.2, enabling the use of non-repeatable partitioning functions without resulting in permanent

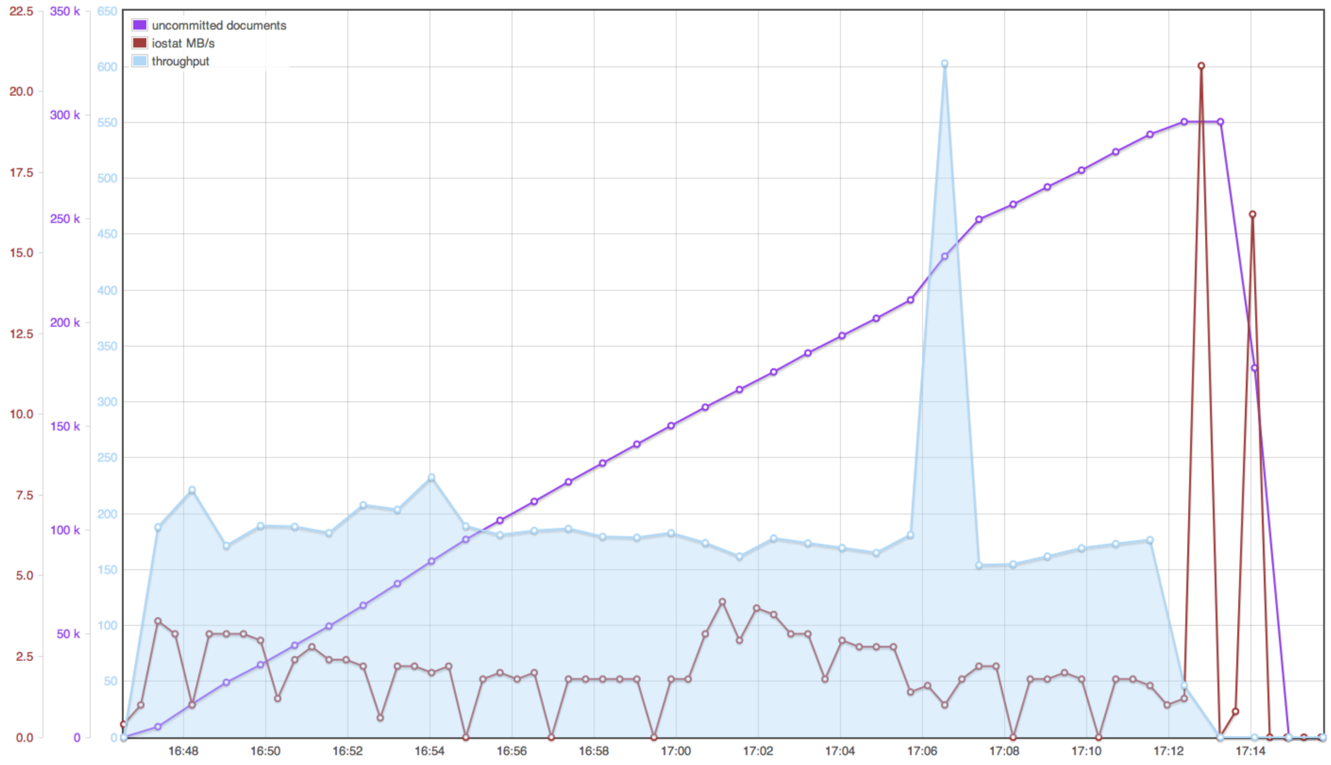


Figure 8.15: Online merging of two partitions with $300k$ documents each. The spike in throughput is the result of a series of short documents that occurs naturally in the WEX document collection.

document duplication by publishing and listening for the required messages all the time, not only during rebalancing.

Method	**Time**
Offline	32m37.66s
Online delete-early	23m25.53s
Online delete-when-queued	**23m17.98s**
Online delete-after-flush	31m36.11s

Figure 8.16: Comparison of different rebalancing strategies by total time taken.

Method	**Time**
Offline	**14m15.96s**
Online	28m28.14s

Figure 8.17: Comparison of offline and online merging by time taken.

The time taken by the offline and online merging of databases is shown in Figure 8.17, and shows that offline merging is significantly faster, almost twice as fast, than online merging. This is the result of the online repartitioning incurring an overhead cost by having to serialize, send, receive and unserialize all the documents that are rebalanced, in addition to not being able to perform the database-specific operations the offline repartitioning does.

Chapter 9

Conclusion

The benchmarks in Chapter 8 show that online repartitioning is competitive to other repartitioning strategies. They also show that tool that perform database-specific optimizations will continue to have a significant edge over the general high-level API when it comes to rebalancing time. Exposing these optimizations to users of the high-level API instead of limiting it to specific tools could potentially result in large improvements in online rebalancing time.

The processing framework introduced in Chapter 7 shows that partitioning, repartitioning and rebalancing is possible to implement quickly. Repartitioning and rebalancing in order to scale in or out depending on the load is a viable strategy that can be supported with very few changes to an existing indexing configuration. This is important in order to minimize the cost of computing in the cloud. Processing frameworks also enable quick exploration of different partitioning strategies which is of great value when determining a partitioning scheme for a specific deployment and is essential in implementing distributed indexing on top of todays full-text index servers such as Solr and Xapian.

9.1 Further work

Due to time constraints, the benchmarks were only executed once. In order to gain more confidence in the benchmark numbers, they should be re-run several times, possibly in several different availability zones and at different times of the day.

Node failures and message-resending should be considered simulated, as they are more likely to occur in a non-controlled environment over time than in the benchmarking environment used in Chapter 8.

In the case of offline-repartitioning, a database specific tool could be written for splitting a database into one or more partitions to achieve the same performance as “xapian-compact”.

The performance under regular query- and continuous indexing load was not benchmarked, which is likely to be a factor in a real-world deployment. Additionally, the impact of the repartitioning on the replicas should be considered, as it might be possible to save synchronization time by performing the repartitioning on some or all of the replicas in concurrently with the master index repartitioning.

Tools that assist administrators in creating and performing automated and custom rebalancing processes could be developed. Libraries such as Fabric [80], Puppet [81] and Chef [82] can be used to enable performing sophisticated administration task on a distributed set of machines with relative ease could be used as a future base for these tools.

Support for performing distributed two-phase commit should be considered written and included in search systems in order to support distributed indexing without consistency anomalies. This is made non-trivial by the existence of replicas, which should be considered to achieve correct distributed consistency.

Partitions of deliberately different sizes could be looked into and possibly supported by a well-crafted partitioning function in order to support relative balancing between nodes of non-uniform sizes.

Distributed file systems should be investigated to see which interesting features they provide for implementing a distributed search system. Large indexes can be difficult to scale up even using distributed file systems, partly because less of the index will fit in the memory of a node, but as a method of distributing the index databases to query nodes it could prove useful.

The use cases for the different partitioning strategies in Section 5.4 should be explored further to get a better overview of which situations one of the partitioning strategies are better suited than the others.

Different messaging systems using different policies (see Section 3.5) and messaging patterns should be compared to see if any of the features they provide can assist in creating more easily maintainable distributed processing systems.

Appendix A

Indexing implementation

In order to give the reader an insight into how the initial indexes were created, this appendix shows how the processing framework were used to create the 8 initial partitions used in the benchmarks.

The partitions were created by eight indexing nodes (see Figure A.1) using configurations that only differ in the database paths used.

Documents from the WEX collection were sent to the partitions by binding an ZeroMQ PUSH queue (see Section 3.5.1) to `localhost:6000` and sending serialized documents to that queue. This resulted in a load-balanced partition between the indexing nodes.

```yaml
zmq:
    queues:
        input:
            pipeline: index
            type: PULL
            sockopts:
                - key: HWM
                  value: 100
            connects:
                - tcp://0.0.0.0:6000

pipelines:
    index:
        # unserialize the document
        - decode-json
        - decode-string:
              input_path: wex_document
              encoding: base64
        - unserialize-xapian-document:
            document_path: wex_document

        # insert the document into the full-text index
        - replace-xapian-document:
              unique_term_path: unique_term
              document_path: wex_document
              index_name: wex
        - set-xapian-metadata:
            key_path: unique_term
            index_name: wex
            namespace:
                now: datetime.datetime.now
            formatter: "baton: dict(index_operation='index', mtime=now())"

    commit:
        - commit-xapian-index:
            index_name: wex

indexes:
    wex:
        locals:
            - !filepath wex.xapian
```

Figure A.1: `index.yaml`: an index node configuration.

Appendix B

Rebalancing implementation

The rebalancing implementation consists of three parts: a source index configuration, a destination index configuration and the rebalancing process. The configuration files are written in YAML [77] while the rebalancing process is written in Python.

```
## source.yaml
zmq:
    queues:
        rebalance:
            pipeline: rebalance
            type: PULL
            sockopts:
                - key: HWM
                  value: 100
            binds:
                - tcp://0.0.0.0:5000

        second_index:
            type: PUSH
            sockopts:
                - key: HWM
                  value: 100
            connects:
                - tcp://0.0.0.0:6100

        # if delete-when-queued or delete-after-flush:
        listener:
            type: SUB
            pipeline: listener
            sockopts:
                - key: SUBSCRIBE
                  value: ''
            connects:
                - tcp://0.0.0.0:7100


pipelines:
    commit:
        - commit-xapian-index:
            index_name: wex

    # if delete-when-queued or delete-after-flush:
```

```
    listener:
        - parse-message:
            # operation mtime unique_term
            formatter: "message: dict(unique_term=message.split(' ', 2)[-1])"

        - delete-xapian-document:
            index_name: foobar
            key_path: unique_term

        - set-xapian-metadata:
            index_name: foobar
            key_path: unique_term
            metadata: '' # setting the metadata to the empty string deletes it

    rebalance:
        - parse-message:
            formatter: "message: dict(unique_term=message)"
        - get-xapian-document:
              unique_term_path: unique_term
              output_path: wex_document
              index_name: wex
        - serialize-xapian-document:
            document_path: wex_document
        - encode-string:
              input_path: wex_document
              encoding: base64

        # if delete-early:
        - delete-xapian-document:
            index_name: wex
            unique_term: unique_term
        - set-xapian-metadata:
            index_name: foobar
            key_path: unique_term
            metadata: '' # setting the metadata to the empty string deletes it

        # encode and send the document to the second index
        - processor: encode-json
        - processor: send-message
          queue_name: second_index


indexes:
    wex:
        locals:
            - !filepath /media/part_01/wex01.xapian
```

The destination index configuration supports publishing messages about queued and flushed documents. Compare the following configuration to the configuration found in Figure A.1 to see the differences between a basic node and a node that publishes messages that could also be used for logging and statistics.

```
## destination.yaml
zmq:
    queues:
        input:
            pipeline: index
            type: PULL
            sockopts:
                - key: HWM
                  value: 100
            binds:
                - tcp://0.0.0.0:6100

        # if delete-when-queued or delete-after-flush:
        broadcast:
            type: PUB
            binds:
                - tcp://0.0.0.0:7100

contexts:
    documents: !!set {}


pipelines:
    index:
        - decode-json
        - decode-string:
              input_path: wex_document
              encoding: base64

        - unserialize-xapian-document:
            document_path: wex_document

        - replace-xapian-document:
              unique_term_path: unique_term
              document_path: wex_document
              index_name: wex
        - set-xapian-metadata:
            key_path: unique_term
            index_name: wex
            namespace:
                now: datetime.datetime.now
            formatter: "baton: dict(index_operation='index', mtime=now())"

        # if delete-when-queued
        - send-message:
            message_path: broadcast
            queue_name: broadcast
            namespace:
                now: datetime.datetime.now
            formatter: "baton: 'queued %s %s'%(now(), baton['unique_term'])"

        - fetch-context:
            context: documents

        # if delete-after-flush:
        - add-to-set:
            set: documents
            formatter: "baton: baton['unique_term']"

    commit:
        - commit-xapian-index:
            index_name: wex

        - fetch-context:
            context: documents

        # if delete-after-flush:
        - broadcast-indexed-messages:
            unique_terms: documents
            namespace:
                now: datetime.datetime.now
            formatter: "unique_term: 'indexed %s %s'%(now(), unique_term)"

indexes:
    wex:
        locals:
            - !filepath /media/part_02/wex02.xapian
```

And finally, the rebalancing process:

```
## rebalance.py
import zmq
import xapian

from_name = '/media/part_01/wex01.xapian'
from_index = xapian.Database(from_name)
total_docs = from_index.get_doccount()

# create a zmq socket connection to the rebalance queue of the source node
c = zmq.Context()
s = c.socket(zmq.PUSH)
s.setsockopt(zmq.HWM, 100)
s.connect('tcp://0.0.0.0:5000')


for i, unique_term in enumerate(from_index.metadata_keys()):

    # sends the unique terms to the rebalance queue
    s.send(unique_term)

    # rebalance half the documents
    if total_docs/2 <= i:
        break
```

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, April 2010.

[2] M. Buckland and F. Gey, "The relationship between recall and precision," *Journal of the American Society for Information Science*, vol. 45, no. 1, pp. 12–19, 1994.

[3] Xapian.org, "Xapian project." `http://xapian.org`.

[4] Apache Software Foundation, "Solr - an open source search platform." `http://lucene.apache.org/solr/`.

[5] 10gen, Inc, "Mongodb." `http://mongodb.org`.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[7] Citrusbyte, sponsored by VMWare, "Redis, an advanced key-value store." `http://redis.io/`.

[8] The Apache Software Foundation, "Couchdb, a document-oriented database." `http://couchdb.apache.org/`.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, (New York, NY, USA), pp. 205–220, ACM, 2007.

[10] Apache Cassandra, "Apache cassandra - a highly scalable second-generation distributed database." `http://cassandra.apache.org/`.

[11] Project Voldemort, "Voldemort." `http://project-voldemort.com/`.

[12] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, pp. 5–, August 2004.

[13] 101tec Inc., “Katta - a distributed storage with real-time access.” `http://katta.sourceforge.net/`.

[14] The Apache Software Foundation, “Hadoop.” `http://hadoop.apache.org/`.

[15] Jake Luciani, “Lucandra- a cassandra backend for lucene/solr.” `https://github.com/tjake/Lucandra`.

[16] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, April 2010.

[17] D. Comer, “The ubiquitous b-tree,” 1979.

[18] D. E. Knuth, “Optimum binary search trees,” vol. 1, pp. 14–25, Jan. 1971.

[19] K. Bratbergsengen, “Kompendium tdt4225 - lagring og behandling av store datamengder,”

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2nd revised edition ed., September 2001.

[21] Apache Software Foundation, “Lucene.” `http://lucene.apache.org/`.

[22] University of Glasgow, “Terrier ir platform.” `http://terrier.org/`.

[23] O. Betts, “The flint backend database structure.” `http://trac.xapian.org/wiki/FlintBackend/Structure`, 2009.

[24] J. Paul Morrison, *Flow-Based Programming*. Nostrand Reinhold, 1994.

[25] J. P. Morrison, “Data stream linkage mechanism,” *IBM Syst. J.*, vol. 17, pp. 383–408, December 1978.

[26] Jody Condit Fagan, “Mashing up multiple web feeds using yahoo! pipes,” in *Computers in Libraries, VOLUME 27, NUMBER 10*, 2007.

[27] M. Altinel, P. Brown, S. Cline, R. Kartha, E. Louie, V. Markl, L. Mau, Y.-H. Ng, D. Simmen, and A. Singh, “Damia: a data mashup fabric for intranet applications,” in *Proceedings of the 33rd international conference on Very large data bases*, VLDB ’07, pp. 1370–1373, VLDB Endowment, 2007.

[28] MuleSoft Inc, “Mulesoft esb.” `http://www.mulesoft.org/`.

[29] The Apache Software Foundation, “Apache camel.” http://camel.apache.org/.

[30] AMPQ, “Amqp specification.” http://www.amqp.org/confluence/display/AMQP/AMQP+Specification.

[31] VMWare, “Rabbitmq.” http://rabbitmq.com.

[32] The Apache Software Foundation, “Activemq.” http://activemq.apache.org/.

[33] iMatix Corporation, “Zeromq.” http://www.zeromq.org/.

[34] Amazon Web Services, “Amazon simple queue service (sqs).” http://aws.amazon.com/sqs/.

[35] iMatix Corporation, “Whitepaper: Broker vs. brokerless.” http://www.zeromq.org/whitepapers:brokerless, 2010.

[36] JSON, “Javascript object notation - json.” http://www.json.org/.

[37] Google, “Protocol buffers - google’s data interchange format.” http://code.google.com/p/protobuf//.

[38] iMatix Corporation, “Zeromq.” http://www.zeromq.org/.

[39] BSON, “Binary json.” http://bsonspec.org/.

[40] W. Hong, “Exploiting inter-operation parallelism in xprs,” *SIGMOD Rec.*, vol. 21, pp. 19–28, June 1992.

[41] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *Information Theory, IEEE Transactions on*, vol. 23, pp. 337 – 343, May 1977.

[42] Twisted Matrix, “Introduction to perspective broker.” http://twistedmatrix.com/documents/10.2.0/core/howto/pb-intro.html.

[43] R. Baeza-yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri, “Challenges on distributed web retrieval,” 2007.

[44] S. Brin and L. Page, “The anatomy of a large-scale hypertextual web search engine,” *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, 1998.

[45] J. Gray, “Why do computers stop and what can be done about it?,” 1985.

[46] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, pp. 382–401, July 1982.

[47] Xapian, "Xapian database replication users guide." `http://xapian.org/docs/replication`.

[48] SOLR, "Index replication in solr." `http://wiki.apache.org/solr/SolrReplication`.

[49] Xapian, "Xapian database replication protocol." `http://xapian.org/docs/replication_protocol`.

[50] D. Puppin, F. Silvestri, and D. Laforenza, "Query-driven document partitioning and collection selection," in *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, (New York, NY, USA), p. 34, ACM, 2006.

[51] Martin Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[52] R. R. Schaller, "Moore's law: past, present, and future," *IEEE Spectr.*, vol. 34, pp. 52–59, June 1997.

[53] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, (Berkeley, CA, USA), pp. 57–70, USENIX Association, 2008.

[54] I. Damgård, "A design principle for hash functions," in *Advances in Cryptology — CRYPTO' 89 Proceedings* (G. Brassard, ed.), vol. 435 of *Lecture Notes in Computer Science*, pp. 416–427, Springer Berlin / Heidelberg, 1990.

[55] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *SIGCOMM Comput. Commun. Rev.*, vol. 19, pp. 1–12, August 1989.

[56] R. W. Dante Salvini, Claudia Dolci and M. Schrattner, "Spatial partitioning and indexing," *Geographic Information Technology Training Alliance*, 2010.

[57] B. S. Everitt, *Cluster analysis / [by] Brian Everitt*. Heinemann Educational [for] the Social Science Research Council, London :, 1974.

[58] A. K. Jain and R. C. Dubes, *Algorithms for clustering data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[59] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer Series in Statistics, New York, NY, USA: Springer New York Inc., 2001.

[60] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, 2002.

[61] D. D'Souza, J. A. Thom, and J. Zobel, "Collection selection for managed distributed document databases," *Information Processing & Management*, vol. 40, no. 3, pp. 527 – 546, 2004.

[62] Xapian, "Uniqueids in xapian." `http://trac.xapian.org/wiki/FAQ/UniqueIds`.

[63] SOLR, "The solr uniquekey." `http://wiki.apache.org/solr/UniqueKey`.

[64] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation or the sun network filesystem," 1985.

[65] Gluster, "Gluster storage platform." `http://gluster.org`.

[66] The Apache Software Foundation, "Hadoop distributed file system." `http://hadoop.apache.org/hdfs/`.

[67] Xapian, "Remote backend protocol." `http://xapian.org/docs/remote_protocol.html`.

[68] D. Smiley and E. Pugh, *Solr 1.4 Enterprise Search Server*. Packt Publishing, 2009.

[69] G. A. Gibson and R. Van Meter, "Network attached storage architecture," *Commun. ACM*, vol. 43, pp. 37–45, November 2000.

[70] D. L. Mills, "Internet time synchronization: The network time protocol," 1991.

[71] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks," *Inf. Process. Lett.*, vol. 43, pp. 47–52, August 1992.

[72] L. Lamport, "Ti clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[73] Twisted Matrix Labs, "Deferreds in depth." `http://twistedmatrix.com/documents/current/core/howto/deferredindepth.html`.

[74] MSDN, "Asynchronous programming for c# and visual basic." `http://msdn.microsoft.com/en-us/vstudio/async`.

[75] Mochikit, "Mochikit.async - manage asynchronous tasks." http://mochi.github.com/mochikit/doc/html/MochiKit/Async.html.

[76] C. T. Haynes, D. P. Friedman, and M. Wand, "Obtaining coroutines with continuations," *Computer Languages*, vol. 11, no. 3-4, pp. 143 – 153, 1986.

[77] I. d. N. Oren Ben-Kiki, Clark Evans, "Yaml ain't markup language (yaml™) version 1.2." http://www.yaml.org/spec/1.2/spec.html, 2009.

[78] M. Braschler and B. Ripplinger, "How effective is stemming and decompounding for german text retrieval?," *Information Retrieval*, vol. 7, pp. 291–316, 2004. 10.1023/B:INRT.0000011208.60754.a1.

[79] M. Technologies, "Freebase wikipedia extraction (wex)." http://download.freebase.com/wex/, 2011.

[80] C. V. Hansen and J. E. Forcier, "Fabric." http://fabfile.org.

[81] Puppet Labs, "Puppet - an enterprise systems management." http://www.puppetlabs.com/puppet/.

[82] Opscode Inc., "Chef - a systems integration framework." http://wiki.opscode.com/.