NTNU
Innovation and Creativity

Automated tuning of MapReduce performance in Vespa Document Store

Knut Auvor Grythe

Master of Science in Computer Science
Submission date: June 2007
Supervisor: Kjetil Nørvåg, IDI
Co-supervisor: Kristian Aune, Yahoo! Technologies Norway
Per Gunnar Auran, Yahoo! Technologies Norway

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

Problem Description

MapReduce is a model for distributed processing, originally designed by Google Inc. VDS is Vespa Document Store, a distributed document storage solution developed by Yahoo! Technologies Norway.

A working prototype of a MapReduce implementation has been developed using Vespa Document Store. However, this prototype is quite immature, and manual tuning of several parameters is required. Most of these parameters affect each other and the system as a whole in a complex manner, and substantial research is required to get a complete understanding of them.

This thesis will focus on researching these parameters, so their effects can be fully known. Where applicable, automated tuning of parameters will also be researched. The research will consist of both theoretical modeling of effects and practical verification of results.

Assignment given: 20. January 2007
Supervisor: Kjetil Nørvåg, IDI

Abstract

MapReduce is a programming model for distributed processing, originally designed by Google Inc. It is designed to simplify the implementation and deployment of distributed programs. *Vespa Document Store (VDS)* is a distributed document storage solution developed by Yahoo! Technologies Norway.

VDS does not currently have any feature allowing distributed aggregation of data. Therefore, a prototype of the MapReduce distributed programming model was previously developed. However, the implementation requires manual tuning of several parameters before each deployment. The goal of this thesis is to allow as many as possible of these parameters to be either automatically configured or set to universally suitable defaults.

We have created a working MapReduce implementation based on previous work, and a framework for monitoring of VDS nodes. Various VDS features have been documented in detail, this documentation has been used to analyse how the performance of these features may be improved. We have also performed various experiments to validate the analysis and gain additional insight.

Numerous configuration options for either VDS in general or the MapReduce implementation have been considered, and recommended settings have been proposed. The propositions are either in the form of default values or algorithms for computing the most suitable setting.

Finally, we provide a list of suggested further work, with suggestions for both general VDS improvements and MapReduce-specific research.

Preface

This master thesis was written by Knut Auvor Grythe as part of a Master's degree at the *Department of Computer and Information Science (IDI)* at the *Norwegian University of Science and Technology (NTNU)* in Trondheim, Norway.

The intention of this project is to research automated tuning of a distributed processing framework running on top of *Vespa Document Storage (VDS)*, a distributed storage solution currently being developed by *Yahoo! Technologies Norway (YTN)*.

During my work in this project, I have been privileged by the supervision of Kjetil Nørvåg, with co-supervisors Kristian Aune, Cyril Banino-Rokkones and Per Gunnar Auran. Their feedback and guidance have been of great help throughout the thesis work.

I would also like to thank the VDS developers, particularly Thomas Fagerlie Gundersen, Håkon Humberset and Jarl Thore Larsen, for their great help and patience while answering numerous more or less elaborate questions.

Finally, I would like to thank all the other employees at YTN, for their debugging help, moral support and interesting discussions over lunch.

Trondheim, June 2007

Knut Auvor Grythe

Contents

Chapter 1

Introduction

1.1 Motivation

In a world of rapidly increasing storage requirements, the need for storage systems of massive size and high performance is emerging. Distributed storage systems provide a way to store enormous amounts of data, while allowing incremental upgrades to improve system performance and storage capacity.

However, such distributed systems are complex, and thus difficult to configure. Often a wide range of configuration options are available, but information about how these affect system performance is sparse or non-existent. Gaining knowledge of how tuning of such options affects system performance could provide opportunities to improve overall system performance without requiring additional hardware.

Vespa Document Store (VDS) is such a distributed system being developed by *Yahoo! Technologies Norway (YTN)*. VDS is intended for use in a multitude of services provided by *Yahoo! Inc.* One use for VDS is as a storage back-end facility for distributed processing, but no distributed processing feature for VDS has previously been available.

MapReduce is a programming model for distributed systems originally developed by *Google Inc.* While the implementation is company confidential, Google has released a paper, [DG04], which generously describes the concept. This allows for alternate implementations of the same principle.

MapReduce consists of two functions: `map` and `reduce`. The `map` function outputs a series of tuples with a name and a value, and the `reduce` function merges all values associated with a common name, returning the merged result. Before the `reduce` function is invoked, data is transferred between computing nodes so that all values associated with a name are located on the same node, ensuring that all values are merged in the same `reduce` call. See Figure 1.1 for an example MapReduce application.

1.2 Goals and Scope

The goal of this thesis is to implement a MapReduce facility for VDS, and to investigate how various configuration options in VDS affect performance in general, and MapReduce performance in particular. Where applicable, methods for automated tuning of VDS will also be researched. Implementation of this automation is however considered outside the scope of this thesis.

Since the primary use-case for MapReduce in VDS is to do aggregation of existing data in production systems, changing the hardware configuration for doing a single computation is normally too expensive. This thesis will therefore primarily consider software settings configurable at runtime. An analysis of the

```
map(String name, String value):
  // name: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1")

reduce(String name, Iterator values):
  // name: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Figure 1.1: Pseudo code for a simple MapReduce program for calculating word frequencies. [DG04]

cost of VDS operations and how they are affected by configuration options will be performed, for the purpose of improved system understanding.

Developing a complete MapReduce library is required for doing realistic experiments. The implementations of both *map* and *reduce* should be combined into a complete library for writing MapReduce applications. This library should allow running multiple passes of MapReduce by storing all output data in VDS.

To allow proper analysis of experiment results, collection of as much data as possible during the course of the experiments are a great advantage. A light-weight framework for collection of such data is therefore necessary. Such a framework should be developed, preferably without unnecessary re-implementation of existing data acquisition software.

This thesis builds upon the work of [Gry06], where several prototypes for a VDS implementation of MapReduce were researched.

1.3 Contributions

In this thesis, various contributions have been made, both in software development and research. We give a brief overview.

A MapReduce library for VDS has been implemented, partially based on previous work. Several applications have been developed using this framework and used in experiments.

A framework for scheduling of test runs used in experiments has also been implemented, including a framework for collecting status information from all cluster nodes. The information collected has been used both for detailed graphs and aggregated information.

Documentation of various VDS features has been created, with help from VDS developers. This documentation has been used for performance analysis associated with experiments.

Numerous experiments have been done to explore the effect of various VDS and MapReduce settings, and default values of algorithms for determining suitable values have been proposed.

1.4 Outline

This section briefly describes the outline of the report.

Chapter 1 contains this introduction.

Chapter 2 describes existing distributed storage systems.

Chapter 3 provides previously unpublished information about VDS.

Chapter 4 describes the Google MapReduce programming model, and the VDS MapReduce implementation.

Chapter 5 researches general VDS configuration relevant to MapReduce.

Chapter 6 describes and researches MapReduce specific configuration.

Chapter 7 lists some suggestions for further work.

Chapter 8 gives an overview of the experiences from the thesis work.

Chapter 9 contains the conclusion.

Chapter 2

Distributed storage systems

This chapter gives a brief introduction to some existing storage systems relate to MapReduce and Vespa Document Store.

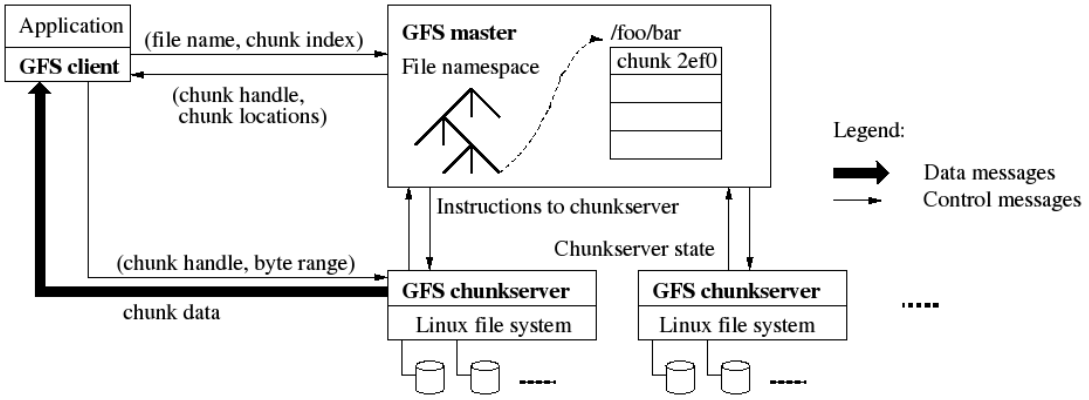

Figure 2.1: An overview of the GFS architecture [GGL03]

2.1 The Google File System

The Google File System (GFS) [GGL03] is a distributed file system designed by Google Inc. It is designed to specifically meet the requirements of the distributed applications at Google.

The files in GFS are split in one or more *chunks*, with each chunk being up to 64 MB large. Chunks are identified by a 64-bit ID, but often addressed by their file name and sequence number. A GFS cluster consists of a single *master* holding the file namespace with pointers to chunks, and multiple *chunkservers* storing them. The cluster is accessed by *clients*. A node may be both a chunkserver and a client at the same time, provided the administrator accepts the added risk associated with running client code on the chunkserver. See Figure 2.1 for an illustration of the GFS architecture.

2.1.1 Assumptions

Based on knowledge of the applications used at Google, the developers of GFS were able to do some assumptions to aid their design. A few of their assumptions are outlined here.

- GFS will run on commodity hardware, and this implicates common hardware failures. It must be able to recover from these failures gracefully.
- The files are few and large. Typically, the system will serve a few million files, and they will often be multiple gigabytes large. Small files should be supported, but the system should only optimise for large files.
- Data in files are usually read in large, streaming reads. If applications need small, random reads, they will sort them sequentially.
- Data is usually appended to files in large writes. Small, random writes should be possible, but not necessarily efficient.
- Multiple clients should be able to append to a file concurrently, typically in a distributed application.
- Files should be usable as producer-consumer queues, with one consumer reading from the beginning while multiple producers are appending to it simultaneously.
- Bandwidth is more important than latency. Most operations transfer a lot of data at once, with transfer time dwarfing the latency.
- Caching of file data is not needed, as data is usually streamed in large chunks.

Name	Meaning
create	Create file
delete	Delete file
open	Open file
read	Read from opened file
write	Write to opened file
snapshot	Copy a file or tree at low cost
record append	Atomically append to a file

Table 2.1: Operations supported in GFS

2.1.2 Client API

GFS provides an interface similar to a normal hierarchical file system. Although similar in some aspects, it does not implement the POSIX interface. GFS provides a limited set of operations, as shown in Table 2.1.

The operations *create*, *delete*, *open*, *read* and *write* should be fairly self-explanatory, and will not be discussed further. *Snapshot* and *record append* however, are less common and thus require some explanation.

Record append appends data to the end of a file. Multiple clients may append data concurrently. Record append guarantees that the data is appended at least once, but does not guarantee that data isn't duplicated. Record append is commonly used for merging data from multiple clients. It is also used to implement producer-consumer queues, where several clients append to a file, and another client reads the file concurrently.

Snapshot creates an almost instant copy of a file or directory tree. This is done by using a technique called copy-on-write, where data is not actually copied before the original data is modified. Google developers commonly use *snapshot* to do branching or create backup copies.

2.1.3 The master node

The master is responsible for keeping the file namespace, the mapping from files to chunks, and the locations of all chunk replicas. All information is kept in memory at all times, to improve performance and avoid I/O on the master becoming a performance bottleneck.

Since the master is the authoritative source of the file namespace and file-chunk mappings, this information is also stored persistently. To avoid costly random writes during updates, all mutations of this information is written to an *operation log*. The operation log is stored both locally and on remote machines, in case of hardware failures.

The master does not keep the location of chunk replicas persistently. Instead, chunk servers report their inventory to the master on startup. Chunk server failures are detected using heartbeat messages, and re-replication of chunks is initiated by the master.

The master is also responsible of controlling the level of replication on all chunks, and for selecting where chunks should be located. Failures affecting whole racks, like failed network equipment, should not take all chunk replicas offline. Also, in the case of hot spots, multiple replicas should not be limited by the same network uplink, but instead balance load across multiple racks. The tradeoff is slower writes, since they must travel across multiple racks as well. This is considered acceptable.

Reasoning behind the single master design

At first glance, having a single master looks like poor design, both from a load balancing and fault tolerance point of view. The reasoning behind this design is as follows:

- Master failures are considered unlikely compared to chunkserver failures.
- Having only a single master simplifies the design.
- It allows the master to make decisions based on global knowledge.
- In the rare occurrence of a master failure, another master may be started using the replicated operation log.

2.1.4 Chunk leases

A primary chunk replica is selected by the master by granting it a *lease* of the chunk. The lease has an initial timeout of 60 seconds, but may be extended via requests piggybacked on the heartbeats sent to the master

In some cases, the master may revoke a lease before it expires, temporarily leaving the chunk without a primary. This has the effect of making the chunk read-only, and is used for example while renaming a file.

If the master loses contact with the chunkserver currently having the lease, it may safely grant a new lease to another chunkserver after the lease timeout expires.

2.1.5 Data flow of operations

GFS is designed to minimise traffic to the master. Thus requests from clients to the master are primarily related to the file namespace alterations and chunk location lookups, while chunk access is done directly to chunk servers. To minimise client-master communication, the client caches chunk locations for a limited time. During this period, all subsequent operations on the same chunks use cached values instead of contacting the master. If the client plans to access multiple chunks, multiple chunk location requests are included in the same request, to minimise the amount of interactions.

File alterations

The data flow of *write* requests in GFS is outlined in Figure 2.2. The client first computes the sequence numbers of all chunks it is going to modify. It then requests the location of these chunks from the master. The master replies with lists of all relevant chunkservers, and identifies which chunkserver is the primary holding the lease.

After receiving the list of chunkservers storing a chunk, the clients transfer the data to one of the chunkservers. Usually, the server closest to the client on the network is chosen, based on IP address. In parallel with receiving data, this chunkserver forwards the same data to the next chunkserver, and so on. The chunkservers do not apply the change at once, but cache the data until it is used or a timeout is reached.

When all replicas have received the update and acknowledged to the client, it sends a write request to the primary replica. The primary replica assigns a sequence number to all pending mutations to the chunk. The same sequence is used for all clients writing to the chunk, ensuring race conditions between clients do not occur. It then applies the changes in the order specified by the serial numbers.

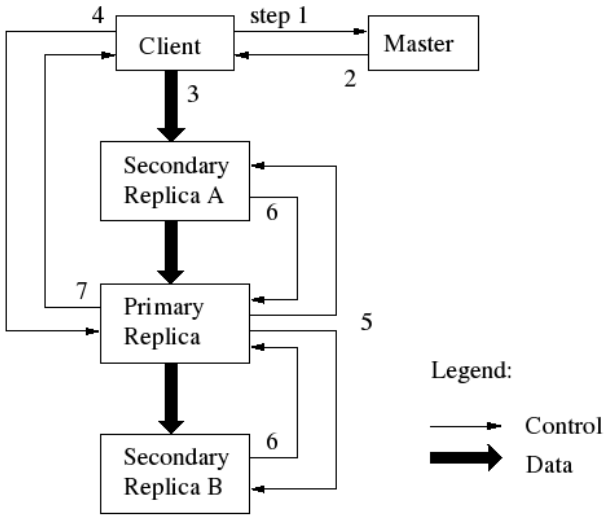

Figure 2.2: Data flow of file alterations in GFS [GGL03]

After committing changes locally, the primary forwards the request to all secondary replicas, along with the correct sequence for mutations. The replicas reply when the changes are applied. The primary then acknowledges to the client.

In case of errors on secondary replicas, the modified region will differ between replicas. In GFS terms, this is an *inconsistent state*. An error message is piggybacked on the reply to the client, which must then repeat the request to overwrite the area with consistent data.

Note that file alteration requests may only apply to one chunk, not files as a whole. This means alternations spanning multiple chunks must be split into independent requests. With clients concurrently modifying multiple chunks of the same file, this may result in modifications on each chunk being performed in different order, leaving the file content in an *undefined state*, with partial data from each client.

Record append

Record append provides an atomic way to append data to files, without risking to render the files in an undefined state. Record appends work in a similar manner as regular file alterations, but adds some extra logic on the primary chunkserver.

After determining the last chunk in the file and pushing data to all replicas, the client issues a request to the primary. The primary checks if the appended data would make the chunk exceed 64 MB, the maximum chunk size. If so, the chunk is padded to 64 MB and the client is asked to retry the request on the next chunk. To avoid excessive amounts of padding in chunks, record appends are limited in size to 16 MB, one fourth of the maximum chunk size.

In case of failures on any of the replica nodes, the client is requested to retry. The data is then appended again, leaving the previously appended data inconsistent between replicas. GFS does thus not guarantee that chunks replicas are byte-wise identical. Inconsistent chunk areas are considered undefined, and GFS clients are capable of handling them.

Snapshot

Snapshot requests are handled in a similar manner as in AFS [HKM+88], using copy-on-write techniques. This implies that a snapshot request completes almost instantly, and that subsequent updates cause the relevant chunks to be copied.

When a snapshot of a file or directory is requested, all leases of relevant chunks are revoked. This ensured that all subsequent writes to these chunks will require interaction with the master. When the affected chunks are later requested for writing, the master may initiate copying of the chunk first. This copying happens locally on the chunkservers, to avoid expensive transfers of chunks across the network.

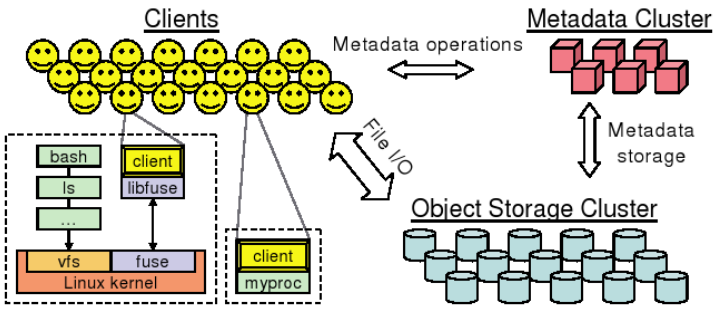

Figure 2.3: An overview of Ceph [WBM+06]

2.2 Ceph

Ceph [WBM+06] is a distributed file system being developed at the University of California, Santa Cruz. It is designed to be massively scalable, and achieves this by storing data in predictable locations, allowing data to be modified without requiring metadata about data locations to be updated.

The Ceph architecture consists of two clusters: The Object Storage Cluster and the Metadata Cluster. The Object Storage Cluster handles storing file contents, while the Metadata Cluster handles all file metadata. Note that the file metadata does not include the location of data on the Object Storage Cluster. These locations are not stored at all, but instead calculated using an algorithm called CRUSH [WBMM06].

When storing a file in Ceph, the file content is striped across several objects with predictable names. These objects are then assigned to some number of storage devices using the CRUSH algorithm. The data used in this algorithm is accessible to any part of the file system, including the client library, and thus allows file data to be located using only an immutable file inode number.

See Figure 2.3 for an overview of the Ceph architecture.

2.2.1 Assumptions

Ceph is designed to be deployed on dynamic systems. It assumes that hardware failures are common, and that nodes may be added, replaced or upgraded when required. It also acknowledges that the popularity of data is going to change over time, since recent data is often accessed more frequently than older data. This is handled by spreading both old and new data evenly across all nodes.

2.2.2 Client API

Ceph provides two interfaces for client applications. One can either link the client library directly into the client application, or mount it as a normal file system using the FUSE [Sze] user-space file system interface. Both alternatives provide the same kernel-independent file data caching.

Client applications are provided with an API similar to POSIX, but with extensions to allow improved performance in certain applications. These extensions include an `O_LAZY` flag for *open* and system calls for manual synchronisation. This allows applications with no need for the strict consistency of POSIX to turn off various atomicity requirements, allowing caching and lazy writes where this would normally not be possible.

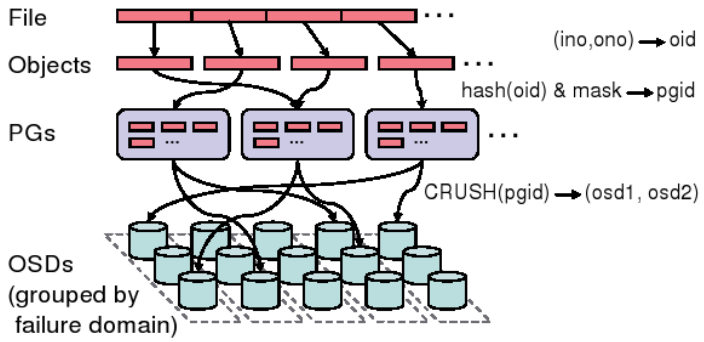

Figure 2.4: How files are mapped to nodes in Ceph [WBM+06]

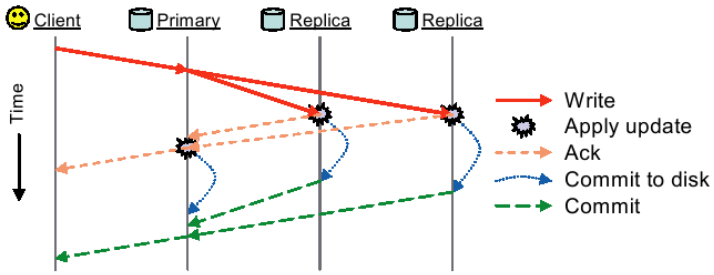

Figure 2.5: How a write in Ceph is performed [WBM+06]

2.2.3 The Object Storage Cluster

The Object Storage Cluster is responsible for all persistent storage in Ceph. It consists of a large number of Object Storage Devices (OSDs) all handling parts of a single, shared namespace.

In order to gracefully handle node failures, Ceph duplicates data across multiple OSDs. Files are striped across many objects, which are grouped into *placement groups (PGs)*. Each placement group is assigned to multiple OSDs using the CRUSH algorithm. This process is illustrated in Figure 2.4.

When the CRUSH algorithm is used on a placement group, it outputs an ordered list of OSDs. The first entry in this list is the primary, while the remaining are replicas. All requests (both read and write) are directed to the primary OSD, which is responsible for syncing file modifications to the replicas.

When a primary OSD receives a write request, it first distributes it to all replicas. When all replicas have applied the update and acknowledged the primary, the primary also applies the write. After acknowledging, all nodes begin committing the change to disk. When finished, replica nodes return a commit message to the primary. The primary completes the local commit and returns an aggregated commit message to the client. This process is illustrated in Figure 2.5.

Synchronous calls on the client normally return as soon as the acknowledgement is received, but written data is per default kept in client memory until a commit message is received. This allows re-submission of data in case of a power failure taking down all relevant OSDs at once.

2.2.4 The Metadata Cluster

Access to file metadata has been shown to amount to about half of typical file system load [RLA00], and Ceph therefore puts great effort in keeping the metadata cluster efficient and scalable. Most notable is

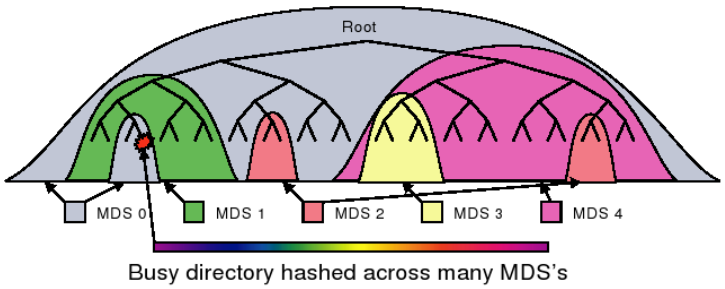

Figure 2.6: Subtree partitioning in Ceph [WBM+06]

the use of dynamic subtree partitioning, a technique for spreading metadata requests evenly across all nodes, but still allowing exploitation of locality. Such a partitioning is illustrated in Figure 2.6.

The metadata cluster uses the object storage cluster for persistent storage. All metadata of files in the same directory are grouped together in the same metadata file. This allows the metadata of all files in a directory to be collected at once, a common real-world scenario. Examples of this include a detailed listing of a large directory.

To avoid numerous scattered writes as metadata is updated, metadata is updated in memory, but not synced to persistent storage at once. Instead, a journal is written. Using a journal eliminates the need for numerous scattered writes by adding cheaper sequential ones, and still provides the safety of persistent storage. This is especially effective in the common case of several consecutive changes to the same metadata, for instance updating the modification time (*mtime*) of a frequently changing file. In the event of a node failure, first reading the metadata from disk and then playing back the journal recreates the state of the failed node.

2.2.5 The CRUSH algorithm

CRUSH is the algorithm [WBMM06] used for controlling the distribution of objects to storage devices in Ceph. It is a pseudo-random algorithm, designed to operate on a heterogeneous, structured cluster. CRUSH does not keep any lists of object locations, but calculates all object locations based on a hierarchic description of the devices in the storage cluster (the *cluster map*), as well as knowledge of the replica[1]

The cluster map

The cluster map is a tree describing the layout of the storage system. The tree consists of two node types, *buckets* and *devices*. Buckets are parent nodes containing one or more buckets or devices, while devices form the leaf nodes of the tree. Figure 2.7 shows an example CRUSH cluster map with three levels: rows, cabinets and disks.

Devices are assigned weights by the system administrator. These weights are used to control the relative amount of data the storage device is responsible for storing. Bucket weights are defined as the sum of the weights of their children. Data utilisation is assumed to be a function of the data size, with the argument that larger devices have more objects, increasing the probability for the device to hold a frequently accessed object. Therefore, these weights are assumed to be sufficient for modelling both amount of data stored and access patterns.

[1]CRUSH is not restricted to selecting replicas, but may also be used for selecting devices to be used in a variety of redundant schemes, like for instance RAID. Refer to [WBMM06] for information on other schemes than replication.

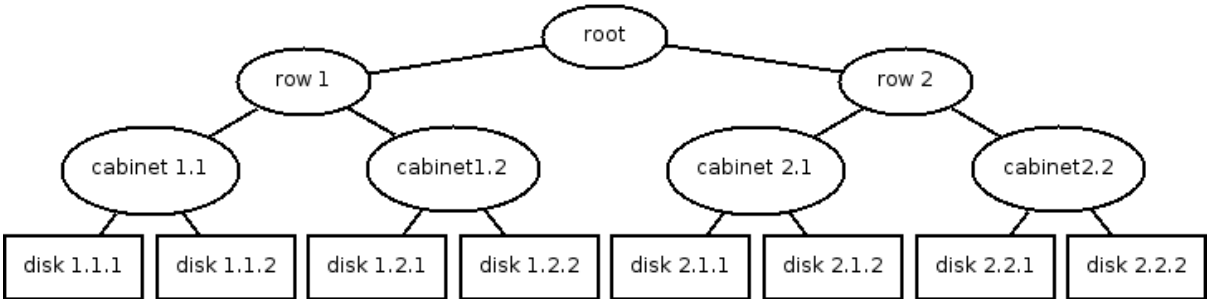

Figure 2.7: An example CRUSH cluster map. Edge weights are omitted for simplicity.

Execution overview

CRUSH is designed to provide a large degree of control over how object replicas are distributed, acknowledging that the optimal distribution of copies is dependent on the individual system. For instance, some systems might desire to locate replicas on different physical locations in case of power outages or natural disasters. Others may want all replicas to be located in the same rack to minimise load on the network backbone. This flexibility is achieved using *placement rules*.

A placement rule consists of several actions to be performed in sequence. When the rule is instantiated, it is provided $\vec{i}$, an empty vector of selected cluster map nodes. For each action in the placement rule, elements in $\vec{i}$ are added or replaced. The resulting vector is a ordered list of devices on which to store the object.

The following placement rule actions are provided:

- `take(node)` – Add *node* to $\vec{i}$.
- `select(n, t)` – Replace each node in $\vec{i}$ with `n` pseudo-randomly selected child nodes of type `t`.
- `emit` – Move all elements in $\vec{i}$ to the final list of storage devices to use.

Below is an example rule for distributing three replicas across three different cabinets in the same row, given a similar (but larger) cluster map as the one presented in Figure 2.7:

Action	**Resulting $\vec{i}$**
`take(root)`	(root)
`select(1, row)`	(row 2)
`select(3, cabinet)`	(cab 2.1, cab 2.3, cab 2.4)
`select(1, disk)`	(disk 2.1.7, disk 2.3.13, disk 2.4.37)
`emit`	( )

More elaborate rules consisting of multiple calls to *take* and *emit* may be constructed, allowing the administrator to specify multiple starting points. This could for example be used to store two local copies and one backup at a remote site:

Action	**Resulting $\vec{i}$**
`take(local root)`	(local root)
`select(2, disk)`	(disk L.2.2.7, disk L.5.3.2)
`emit`	( )
`take(remote root)`	(remote root)
`select(1, disk)`	(disk R.5.2.3)
`emit`	( )

Note that the above example does not specify any rows or cabinets. This will cause CRUSH to descend

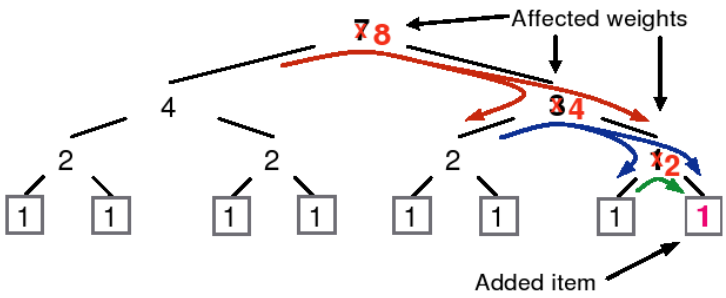

Figure 2.8: Data movement due to a node addition [WBMM06]

Action	Uniform	List	Tree	Straw
Speed	O(1)	O(n)	O(log n)	O(n)
Additions	poor	optimal	good	optimal
Removals	poor	poor	good	optimal

Table 2.2: Bucket types in CRUSH [WBMM06]

recursively until a node of the requested type is reached. The result gives two disks in pseudo-random rows and cabinets.

Node additions and removals

When nodes are added or removed from the cluster map, bucket weights are recalculated. Because the bucket weights affect the assignment of objects to nodes, this causes some of the objects to be moved. An illustration of this process is provided in Figure 2.8.

While some data should be moved to the new node to achieve the proper load balancing, unnecessary data movement should be minimised. The degree of movement when adding or removing nodes depend on the hashing algorithm. CRUSH provides several choices by offering different bucket types, each of which providing a different hashing method.

Bucket types

While `select(n, t)` traverses the cluster map, hashing is done independently for each level. This allows using different hashing algorithms on a per-node basis. CRUSH implements this by allowing the administrator to choose between different bucket types[2] when creating the cluster map.

The various bucket types and their properties are shown in table 2.2, including ratings or the amount of data transferred when nodes are added or removed. When choosing between node types, the administrator should consider the expected degree of node additions and removals, compared to the cost of computations.

Uniform buckets require all their children to have equal weight, and simply choose the appropriate child node by using a predictable hash value modulo the amount of children. In the case of nodes being added or removed, most of the data will hash to a new item, causing a lot of unnecessary data transfer.

[2]Bucket and node types should not be confused. Bucket types are part of the CRUSH algorithm, while node types are names assigned by the administrator

List buckets keep all children in a linked list. CRUSH begins at the first item and chooses whether to use the item or continue based in a hash value, the weight of the current item and the weight of all remaining items. If the item is not chosen, the process continues recursively to the next item. When adding new nodes, this results in values either staying where they are or being shifted to the right in the list, causing an optimal amount of data to travel between nodes. Node removals will however cause a lot of data transfer, so this bucket type is most suitable for clusters where nodes are rarely removed.

Tree buckets keep items as leaf nodes in a binary tree. Child nodes are selected recursively using hashing at each level until reaching a leaf node. This gives fewer iterations compared to tree buckets and less movement when nodes are removed, at the cost of slightly more migration as the cluster is expanded.

Straw buckets use a process similar to a draw of straws to select a bucket. The item with the longest straw wins. The length of each straw is computed using a hash value, and then modified according to the weight of the item. This is the slowest bucket type, on average twice as slow as list buckets. However, it results in optimal data movement in case of both additions and removals of nodes, and is thus suitable for frequently changing clusters.

Chapter 3

Vespa Document Store

Vespa Document Store (VDS) is a distributed storage solution developed by Yahoo! Technologies Norway. It is designed to store large amounts of small data structures called "documents". VDS is not a file system and does not have a directory structure, but instead resembles an object store. It addresses documents by their unique ID.

Documents consist of data fields of various types, specified by the application-specific document types. For an example document type definition, see Figure 3.1.

3.1 Assumptions

When VDS was designed, a few assumptions about the target environment were made. These assumptions are:

- All data will be stored as documents, each with a unique Document ID.
- Documents will be of limited size, so multiple documents could fit in memory.
- VDS will be used in conjunction with a search engine, and should be able to provide documents for indexing, both all at once and incrementally as new documents are added.
- VDS will run on commodity hardware with frequent failures, and should be able to recover from these failures gracefully and without loss or corruption of data.
- The order in which documents are returned is not important.

```
document person {
    field fullname type string {
    }

    field age type int {
    }
}
```

Figure 3.1: An example document type definition.

3.2 Guarantees

VDS provides a few guarantees to programmers. If these guarantees do not suffice, additional requirements must be implemented at application level.

- If a document is stored with a Document ID that already exists in the system, the older document will be overwritten.
- Documents are returned to clients *at least once*. If the client cannot accept duplicates, it must filter these out by itself.
- VDS maximises I/O performance by spreading reads over as many storage nodes as possible, by reading different buckets from different nodes if possible.
- If a server-side visitor succeeds, it will have read all selected data from the buckets it visited. If a node reads a bucket partially and then fails, the entire bucket will be re-read by another node.
- VDS detects hardware failures and attempts to preserve affected data, using replication and checksumming.

3.3 API overview

VDS provides three data APIs: The *document* API takes care of handling of individual documents, the *visitor* API is used for batch jobs on multiple documents, and the *subscription* API monitors changes to the data in VDS.

3.3.1 The document API

The document API is responsible for adding, removing and fetching individual documents. For this purpose, the operations *put*, *remove* and *get* are provided.

3.3.2 The visitor API

The visitor API provides tools to access all or some of the stored documents. A default visitor for returning the visited documents unmodified is provided, and an API for implementing other visitors exists. Custom visitors may for instance be used for preprocessing the documents in some way before they are returned.

Which documents to visit may be selected by using a filter language similar to the WHERE clause in SQL. Both the Document ID and other document fields may be matched, either completely or using wild cards. Additionally, the visiting may be limited to specific buckets.

The visitor API consists of two parts: One client-side visitor, responsible for receiving the data, and one server-side visitor which will run on the individual nodes where the data is stored (see Figure 3.2). Both the server-side and client-side visitor may be replaced with custom code, possibly modifying the results before they are passed on.

When visiting, data is grouped by bucket, causing all data in a bucket to be handled by the same visitor. This allows aggregating data from all documents in a bucket. The visitor may handle a configurable amount of buckets at a time, but which buckets are handled together is undefined. For more details on this grouping, refer to Section 3.6.4.

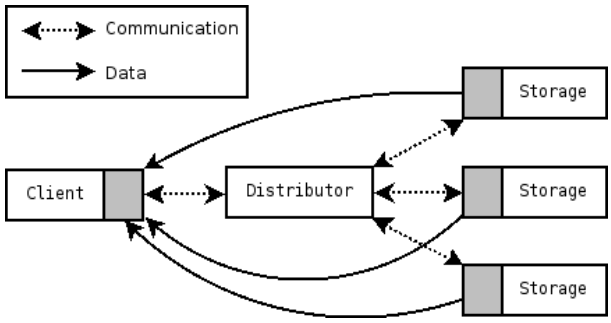

Figure 3.2: The visiting API consists of one client-side and one server-side visitor, both marked in grey.

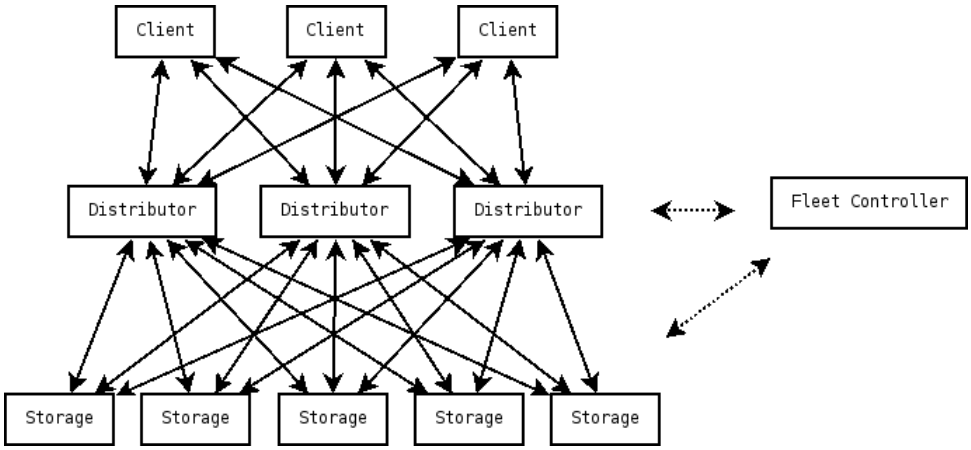

Figure 3.3: The VDS architecture

3.3.3 The subscription API

The subscription API provides an interface for processing the stream of updates to the storage system. Apart from this, it is quite similar to the visitor API. It supports the same selection criteria, and may also be used for preprocessing the documents before returning them.

The subscription API can for instance be used for updating the index of a search engine as new documents are stored or removed.

3.4 Node types

VDS consists of three kinds of nodes: *storage* nodes, *distributor* nodes, and the *fleet controller* node. The relationship between these nodes is illustrated in Figure 3.3.

The storage nodes store the actual data in VDS. Each storage node holds a group of buckets, each containing documents. If fault tolerance is desired, which is the general case, each bucket is stored on at least two storage nodes. The exact number of copies is configurable.

VDS storage nodes do not provide specialised handling of frequently accessed data. Instead, frequently accessed data are assumed to be cached by the underlying operating system.

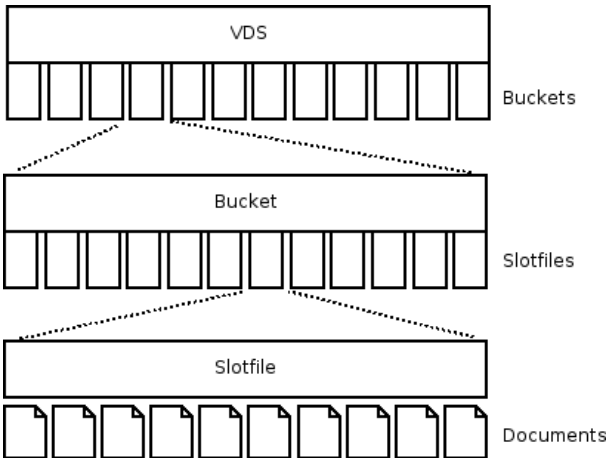

Figure 3.4: The VDS bucket/slotfile allocation hierarchy

The distributor nodes are responsible for managing the distribution of buckets on storage nodes. Distributor nodes collect the current distribution from the storage nodes on startup. They store this distribution in memory, and updates it whenever the distribution changes. They also determine the ideal distribution of buckets given the current set of storage nodes, and continuously work to modify the current distribution towards the ideal by moving and copying buckets between nodes. The algorithm used by the distributor nodes is based loosely on the RUSH family of algorithms [HM, HM03, HM04]. RUSH is a predecessor to the CRUSH algorithm presented in Section 2.2.5, using a flat cluster map instead of a tree.

The fleet controller is responsible for synchronising the state between distributor nodes. All distributor nodes should agree on the global state, as this state affects their view of both the current and ideal distribution of buckets.

The fleet controller maintains a consistent system state by constantly polling the distributor and storage nodes for their current state. A copy of the global state is piggybacked on the request, providing the nodes with an updated state.

The fleet controller is however not responsible for providing the clients with the system state. Client retrieve their state from the distributors, either by polling for it or piggybacked on error messages.

3.5 Buckets and slotfiles

Documents are grouped into *buckets*. A bucket is a form of data unit. Buckets are considered to be atomic, and will never be split across nodes. When data is duplicated on multiple nodes, it is done by copying entire buckets. This design makes reading multiple documents from a single bucket faster than reading them from multiple buckets. The Document ID determines which bucket a document belongs to. If a bucket ID is not specified directly with the use of a specially formatted Document ID, it is determined by hashing the entire Document ID string.

Each bucket is divided into multiple files, called *slotfiles*. These slotfiles have a configurable size, by default 10MB. If a slotfile grows beyond this size, it is split in two. The slotfile size should be configured to be significantly larger than the expected size of stored documents, as multiple documents should fit in each slotfile. See Figure 3.4 for an illustration of the relationship between buckets, slotfiles and documents.

3.5.1 How documents are mapped to buckets

A bucket is selected based on the document ID. Depending on the form of the ID, different selection schemes are used. Currently, the schemes `doc` and `userdoc` are available.

The `doc` selection scheme uses a pure hash of the document ID to select a bucket ID, while the `userdoc` scheme allows specification of an explicit bucket. The latter is especially useful when storing user data, because it allows correlated data to be stored in the same bucket, making retrieval or processing more efficient.

3.5.2 The structure of a slotfile

Before diving into the implementations of the individual commands, the format in which documents are saved on disk should be introduced briefly. In VDS, buckets are stored in the form of one or more files called *slotfiles*. These files contain one or more documents, plus additional metadata.

Slotfiles consist of four sections:

- The slotfile header, where some global file information is stored
- The metadata list, a list of constant size with hashes of document IDs and pointers to document data
- A blob of document headers
- A blob of document bodies

The document headers and bodies both consist of document fields, and whether fields should be a part of the header or body is specified in the document specification. Only the document headers are available for comparisons when retrieving a group of documents. This separation is an optimisation to allow large data to be omitted when documents are compared to a selection string.

When a slotfile is created, it is created with a configurable amount of dummy data. As more documents are added, this dummy data is overwritten with documents. This reduces the fragmentation of the slotfiles in the file system. When the slotfile no longer has room for the next document, it is enlarged, creating more dummy data in it. If the slotfile size reaches a configured limit, it can no longer be resized. In this case, a slotfile split must be performed. A slotfile split is done by splitting all slotfiles in the bucket in two. As a consequence, the number of slotfiles is always a power of two.

When saving a document in a bucket, a unique slotfile is chosen based on some function[1], much like the bucket itself is chosen based on the document ID. This technique of hierarchical allocation is similar to the one presented in [WBMM06], and was illustrated in Figure 3.4. As a consequence, VDS knows exactly which slotfile that holds a certain document, and does not need to search for the document in multiple files.

3.6 Data flow of operations

The document and visitor APIs provide the operations *Put*, *Remove*, *Get* and *Visit*. We give a more detailed explanation of each, with an overview of how calls are executed.

[1] Currently the slotfile is selected using a pure hash of the document ID, but refinements considering document size are planned.

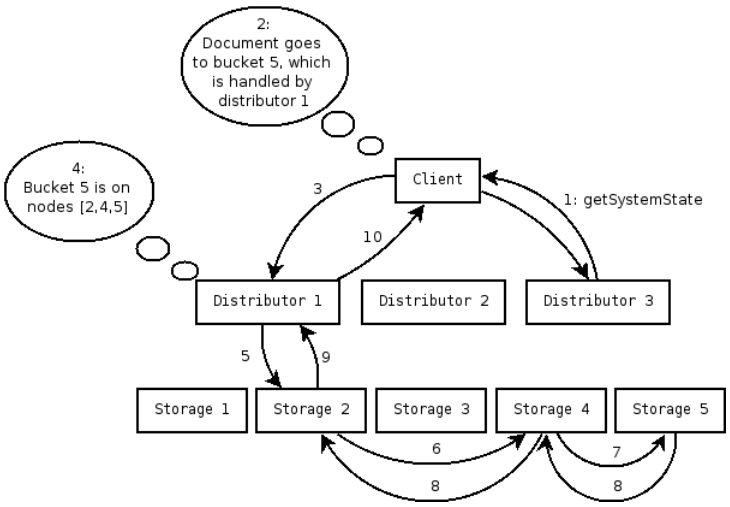

Figure 3.5: Data flow of Put and Remove in VDS

3.6.1 The Put operation

A Put is the only way to enter documents into VDS. Each Put contains only one document, meaning that all documents are added in separate requests.

The order of events in a VDS Put is described below. The enumeration corresponds to the numbers on Figure 3.5.

1. Unless the system state is already known, the storage client picks a random distributor and submits a `getSystemState` command. The distributor replies with the system state, and thereafter the client knows which distributors are responsible for which buckets.
2. The client computes the bucket ID which corresponds to the document ID using a hash function, and looks up which distributor is responsible for the bucket.
3. The client passes the document to the appropriate distributor.
4. The distributor receives the document in its entirety, and looks up which storage nodes contain the appropriate bucket. This results in a sorted list of storage nodes.
5. The storage node forwards the request to the first node in the list (the *primary* node).
6. The primary node receives the document, passes it to the next node replicating the same bucket, and simultaneously starts writing its own copy.
7. The next node repeats the process of forwarding the request (if applicable) and storing the data.
8. Replies are sent back to the previous node.
9. When the last reply has been received by the primary storage node, a consolidated reply is sent to the distributor.
10. The distributor forwards the reply to the client.

When the Put is to be committed to disk, a slotfile is selected based on a hash of the document ID, and the process goes on as follows:

- Read the slotfile header (and possibly some trailing data) from disk.
- Read and iterate over the metadata list until an unused entry or end of list is found.
- Examine the read metadata and confirm that there is available space in the slotfile for the document. If not, space must be made available.
- Write the new document header, body and metadata list entry to file.

If the bucket doesn't exist

Buckets are created on demand, triggered by the first document addition. In the case of a non-existing bucket, the distributor assigns a set of storage nodes for the bucket by using an algorithm similar to the RUSH [HM] family of algorithms. A `createBucket` message is then passed to the primary node, which propagates the request to the other nodes. This causes all relevant nodes to become aware of the new bucket they are responsible for. No files are created until a Put is issued to the bucket later.

The storage nodes handle requests sequentially, so the distributor does not have to wait wait until the bucket is created before the document can be stored. Instead, the Put is submitted shortly after the `createBucket` call. The storage nodes will handle the requests in the correct order.

In the case of inconsistent system state

Node additions and removals from the system will initially cause disagreements about the system state among the nodes. This will eventually be corrected by the fleet controller, but in the meantime the system state will be inconsistent. Cases of inconsistent state are discovered by validating that the request has passed through the correct distributor. This validation is done both at distributors and storage nodes:

1. If a distributor receives a document for a bucket it is not responsible for, it will respond with the *WRONG_DISTRIBUTION* error code, claiming the client has an outdated system state. A copy of the system state is piggybacked on the reply.
2. Similarly, if a storage node receives a document from a distributor it does not believe should handle that bucket, it will respond with an error code, claiming that both the distributor and client has an outdated system state. The error is propagated to the client, which has to retry the request. In the meantime, the distributor will likely have received an updated system state from the fleet controller, and will handle the document correctly.

3.6.2 The Remove operation

Removing documents from VDS is done by writing a new entry in the metadata list, stating that the document is deleted. This makes Remove identical to a Put request (see section 3.6.1), except that no document headers or body is included in the request or the resulting slotfile.

3.6.3 The Get operation

Get is used for fetching a single document from VDS to the client. The steps in the execution are listed below, and the numbers correspond to the numbers on Figure 3.6.

1. Unless the system state is already known, the storage client picks a random distributor and submits a `getSystemState` command. The distributor replies with the system state, and the client now knows which distributors are responsible for each bucket.

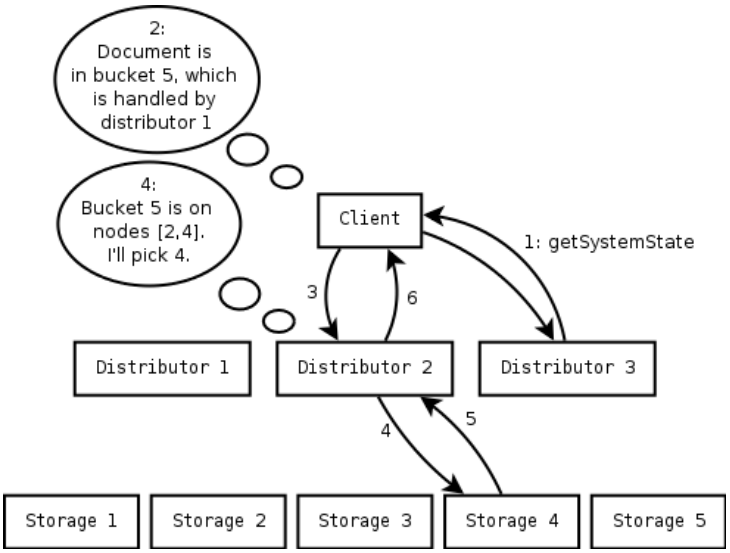

Figure 3.6: Data flow of Get in VDS

2. The client calculates the bucket ID which corresponds to the Document ID using a known function.
3. The client passes the request to the appropriate distributor.
4. The distributor forwards the request to a random[2] updated storage node containing the correct bucket.
5. The storage node replies with the document, and the distributor receives it in its entirety.
6. The distributor forwards the document to the client.

3.6.4 The Visit operation

Visit is the functionality for fetching or processing several documents in batch.

First, the client builds an overview of storage nodes containing relevant buckets:

- Unless the system state is already known, the storage client picks a random distributor and submits a `getSystemState` command. The distributor replies with the system state, and the client now knows which distributors are responsible for each bucket.
- The client groups the buckets by distributor, and issues a `getBucketNodes` call to each distributor with relevant buckets.
- The distributors reply with one updated storage node for each bucket. Each node is chosen randomly from the nodes storing the bucket.

After determining which nodes to use for each bucket, the client creates a configurable amount of visitors on each relevant storage node, and starts receiving data. Each visitor is responsible for a separate subset of the buckets, allowing concurrent processing. The lifespan of a Visitor is described below.

1. The client issues a `createVisitor(buckets[])` call on each relevant storage node, where `buckets[]` is a list of one or more buckets.

[2]More sophisticated load balancing may be added in the future

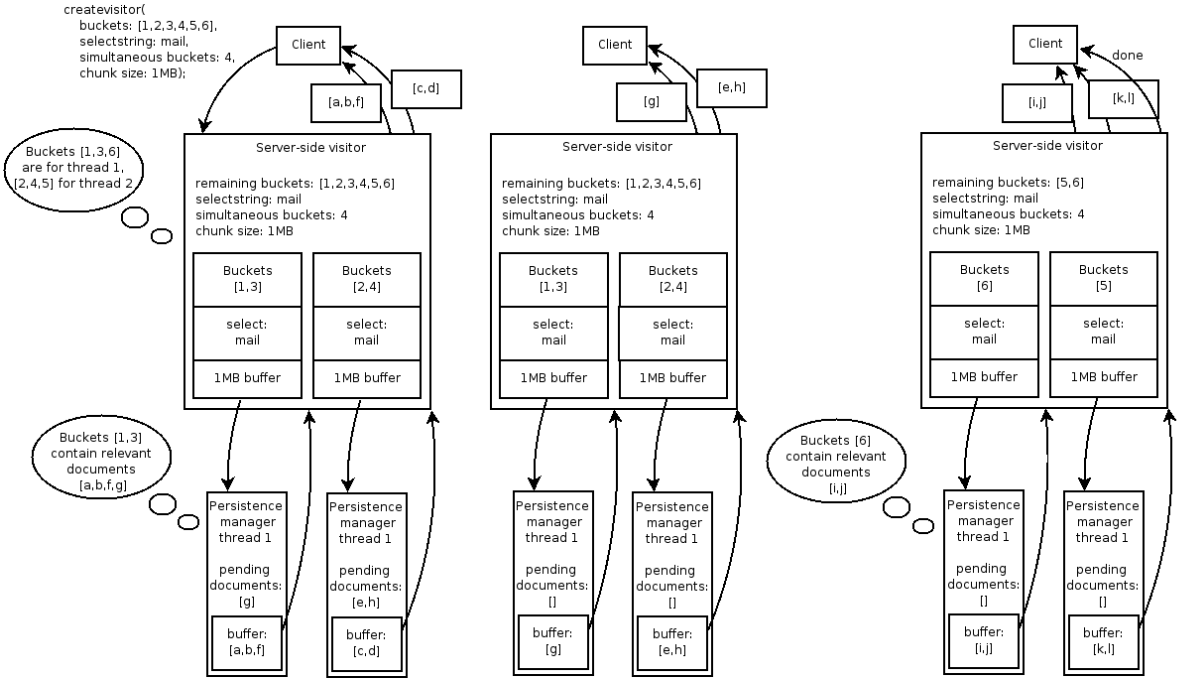

Figure 3.7: The server side visitor visiting documents *a* to *l* in buckets *1* to *8*

2. The storage nodes reply with the ID of the newly created visitor.
3. The storage nodes start locating documents, and pass them back to the client in chunks (more on this below).
4. The client confirms each chunk consecutively.
5. When the last data has been passed back to the client, the storage node submits an End Of File message.
6. The client confirms, and the visitor is destroyed.

On the storage node

After getting a `createVisitor` call, the storage node creates a visitor. The visitor has the following configuration items:

- Which buckets to visit
- A document selection string
- How many buckets to visit simultaneously
- The maximum size of data chunks to be returned (typically 1MB)

The visitor picks the first buckets to visit, and creates a buffer for each bucket. The buffer size is the maximum chunk size. These buffers and the document selection string are passed to the persistence manager.

The persistence manager has a constant (but configurable) number of threads which handle disjoint groups of buckets. Requests for a given bucket will thus always be submitted to the same thread. No effort is made to ensure that multiple buffers are not sent to the same thread at once, as the requests are assumed to spread quite evenly by themselves.

When the persistence manager receives a selection string and buffer, it opens the slotfiles and parses the headers. It adds the documents matching the selection string to a list of document to fetch. It then reads documents from the slotfiles into the buffer until all documents have been fetched or the buffer is full. As documents are put in the buffer, they are removed from the list of pending documents.

When the document buffer is full, it is returned to the visitor via a callback. The visitor is also informed whether all documents have been returned or not. The visitor handles the documents, possibly returning them to the client.

If all documents have not yet been visited, the visitor creates a new buffer and passes it to the persistence manager. The persistence manager reviews its list of pending documents and starts filling the buffer with documents, removing documents from the list as they are added.

As soon as one of the buckets finish, the visitor will create a new buffer and ask for a new bucket. This way, the number of buckets currently being visited is the same until no more buckets are pending for that thread.

The visiting process is illustrated in Figure 3.7.

Chapter 4

The MapReduce programming model

This chapter describes the MapReduce distributed processing framework, and the VDS implementation of it.

```
map(String name, String value):
  // name: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1")

reduce(String name, Iterator values):
  // name: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Figure 4.1: A simple MapReduce program for calculating word frequencies. [DG04]

4.1 Google's MapReduce

Google's MapReduce is a framework for distributed processing, designed to enable developers to write powerful, distributed programs, without requiring them to handle the coordination of a distributed system. The developer simply provides two functions, and the framework is responsible of running these functions in a distributed manner.

4.1.1 Motivation

Distributed processing implies significantly increased complexity compared to centralised processing. Google experienced that the pitfalls of distributed programming was being a bottleneck in their development process, as developers had to create large and complex systems for every distributed task, making them re-invent the wheel instead of focusing on the original problem.

To assess this problem, Google developed MapReduce. It was designed to hide the complexity of distributed processing, and create a clean and simple interface that was easy to understand. This way, developers with little or no experience with distributed programming could perform massively parallel computations without having to consult experts in distributed processing.

4.1.2 User interface

The MapReduce API consists of two functions the developer must overload: `map` and `reduce`. Those familiar with the Lisp programming language might recognise their semantics from there. The function `map(values[])` creates a list of tuples consisting of a name and a value. The MapReduce library then groups all tuples with the same name on common computing nodes, and runs the `reduce(name, values[])` function with a name and all the values associated with it. The output of `reduce` is a single value, computed by aggregating the input values in an application-specific way. After `reduce` has been called on all distinct names and their associated values, the computation is considered done, and the result data is the names and values returned by the `reduce` calls.

Pseudo-code for an example MapReduce program for counting word frequencies is included in Figure 4.1. This example will be further explained in section 4.1.3.

4.1.3 Execution overview

The MapReduce architecture consists of a single master node, and a substantial set of worker nodes. The worker nodes are assigned work by the master, which is responsible for controlling the process. We give a brief description of the data flow during execution, as illustrated in Figure 4.2.

- The data, which is stored in GFS[1] is split into M parts, one part per *map* node. The size of parts is typically 16 to 64 megabytes.
- The map nodes read the data, run the `map` function on it, and stores it on local disk.
- As the *map* output is stored, it is split into R parts, one part per *reduce* node.
- When the *map* nodes have processed sufficient data, they inform the *master* node about where the data is stored.
- The *master* node signals the *reduce* nodes, and tell them where to fetch their data.
- The reduce nodes copy their part of the data from each map node, groups the data by name, and runs the `reduce` function on each distinct name and its associated values.
- Each *reduce* node saves its output data back into files in GFS, with a separate file for each node.
- The MapReduce call finished, and the data may be used, possibly as input data for another MapReduce call.

Example Use-Case: Word frequencies

Suppose you have a document collection of one document, containing these words: ("how", "much", "wood", "would", "a", "woodchuck", "chuck", "if", "a", "woodchuck", "would", "chuck", "wood").

You define a function `map`, which puts each word in a tuple together with the number "1". Some words (like "wood") will be listed in several tuples. You then define a function `reduce`, which takes a single name and a list of values, and calculates the sum of values. It then returns the name and the accumulated value. Pseudo-code for functions is provided in 4.2.

After receiving the results from `map`, MapReduce sorts the tuples by name, and runs the function `reduce` with one name and an associated list of values. The result is a list of (word, count) tuples.

See Figure 4.3 for an illustration of the data at the various stages.

4.1.4 Model limitations

Like any other abstraction, MapReduce trades some flexibility for ease of use. This means that not all problems that can be solved in a distributed system are efficiently solvable using MapReduce, since MapReduce restricts the programmer to a specific model. This model forces the user to create programs that do not require synchronisation between nodes, and require that nodes can work on whatever piece of data that is supplied to them. [FMS+06]

While this model is well-suited for problems where a large amount of independent data is to be aggregated, it is less suitable for other distributable problems, like for instance problems where large amounts of interdependent computations are performed on a relatively small set of data. An example of such a problem is the travelling salesman problem, where several distributed implementations exist. [Pet90, PM90]

[1] GFS, the Google File System, is a distributed file system developed and used by Google, Inc.

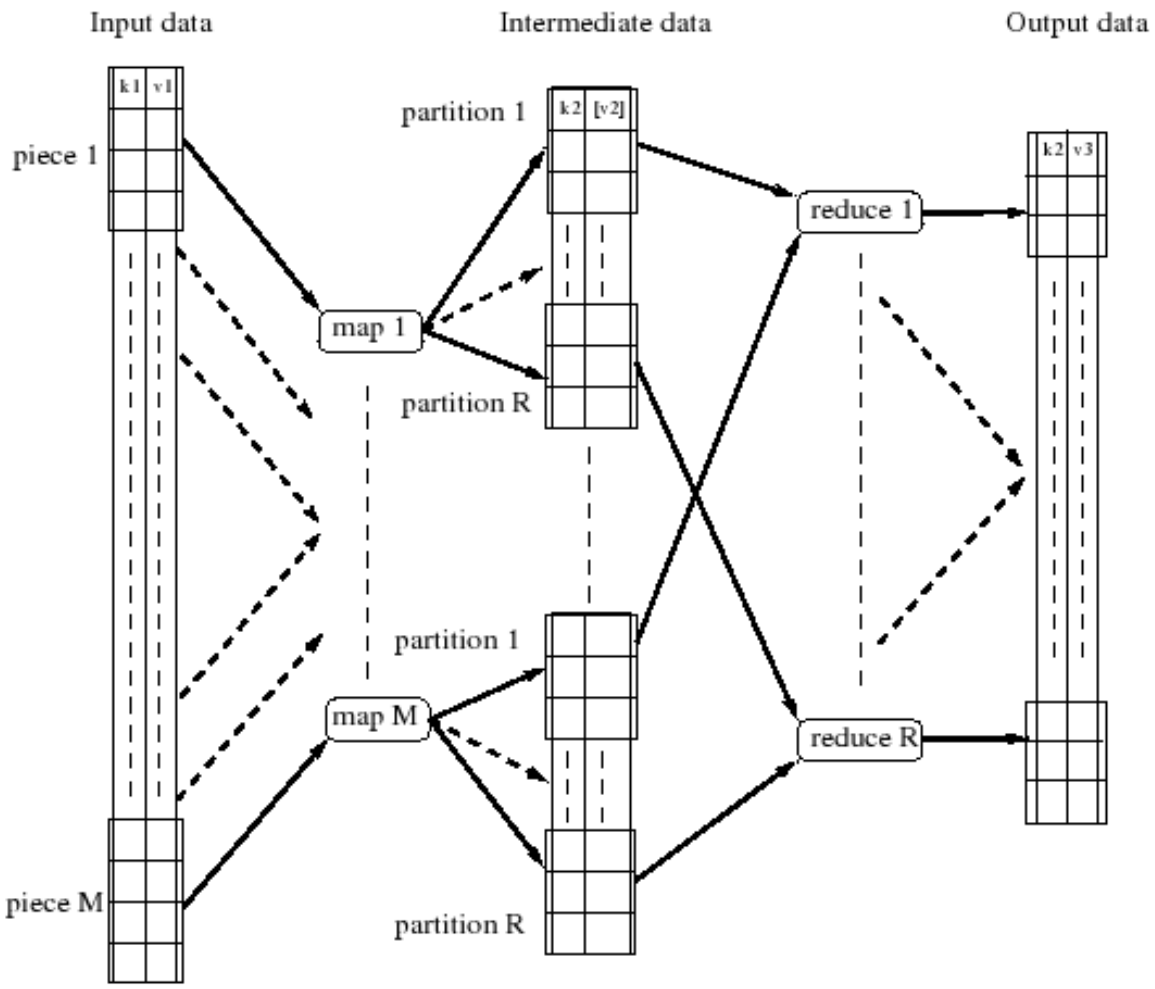

Figure 4.2: MapReduce execution overview [Läm06]

Input data	**Intermediate data**	**Output data**
“how”	(“how”, 1)	(“how”, 1)
“much”	(“much”, 1)	(“much”, 1)
“wood”	(“wood”, 1)	(“wood”, 2)
“would”	(“would”, 1)	(“would”, 2)
“a”	(“a”, 1)	(“a”, 2)
“woodchuck”	(“woodchuck”, 1)	(“woodchuck”, 2)
“chuck”	(“chuck”, 1)	(“chuck”, 2)
“if”	(“if”, 1)	(“if”, 1)
“a”	(“a”, 1)	
“woodchuck”	(“woodchuck”, 1)	
“would”	(“would”, 1)	
“chuck”	(“chuck”, 1)	
“wood”	(“wood”, 1)	

Figure 4.3: Data format in various stages of the word frequency example

4.1.5 Multiple passes for complex calculations

The range of problems solvable with MapReduce may be extended by running it in multiple passes. This way, algorithms involving multiple steps may be implemented, with output from the previous step used as input to the next. [FMS+06]

This method provides an opportunity to run quite complex algorithms using MapReduce. An example of such a complex algorithm is distributed encoding and decoding of text encoded with self-correcting Tornado codes, with a block length too large for a single computer to handle. [Fel06]

4.1.6 Fault tolerance

Fault tolerance in MapReduce is basically achieved by rescheduling subtasks if a node goes down. The system is designed so that only the work done by the particular failed node is lost.

Map worker failure: If a worker fails while doing a map task, the master eventually notices that the node no longer replies. It then reschedules all work originally assigned to that node to other nodes. The already completed work on the node is lost, and must be calculated again.

Reduce worker failure: If a worker fails during a reduce task, work is rescheduled in a similar fashion as when a map worker fails. The task is rescheduled to one or more other nodes, and these have to copy the data from relevant map nodes and re-execute the task.

Master failure: Since there is only one master, it's failure is unlikely. For this reason, MapReduce does not handle master failures. If master failure was to be handled, it would most likely be implemented by writing state checkpoints periodically, and a new master would use the last saved state.

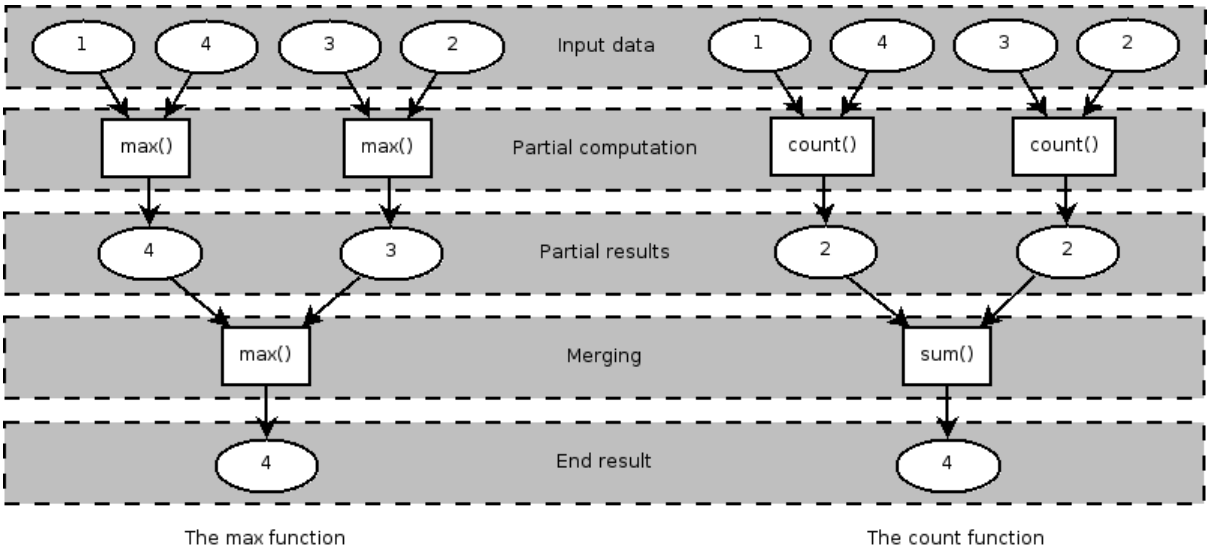

Figure 4.4: Data flow for parallel version of the *max* and *count* functions

4.2 Database-like aggregation using MapReduce

A plausible MapReduce use-case, especially in a VDS context, is the use of MapReduce for distributed aggregation, similar to the aggregation known from database systems. We give a brief introduction to such aggregation and how it can be executed in parallel in distributed databases, before discussing how some of these approaches may be implemented using MapReduce.

When talking about aggregation in database systems, we refer to functions that compute a single values from a group of entries in a database. To tell which values to aggregate, a grouping statement must be provided. See table 4.1 for a list of some common functions for aggregation. [GMUW02]

Count	Count the values
Max	Find maximum value
Min	Find minimum value
Average	Compute average value
Sum	Compute sum of values

Table 4.1: Common aggregate functions in database systems

Most of these functions could be implemented by iterating over all the values and update a single variable. After completion, the variable will contain the end result. An exception is the *average* function, where two values (`sum` and `count`) are updated, and the end result is `sum/count`. As we can see, *average* is simply a combination of *sum* and *count*, and we will therefore not discuss it's differences from the others further, but simply assume it will be implemented using *sum* and *count*.

The *max*, *min* and *sum* functions are all both associative and commutative. They may receive the entries in any order, and multiple partial results may be merged into one by running the same function on the resulting values. This property makes these functions well suited for parallel processing. The *count* function has similar semantics, but partial results must instead be merged with the *sum* function. Figure 4.4 illustrates this property for the *max* and *count* functions. The *min* and *sum* functions behave in a similar manner. Note that although the figure shows one level of merging and two input values for each function, both the number of merge levels and input values may be increased at will. [BBDW83]

As mentioned, aggregation functions are used in conjunction with grouping operations. The grouping operation usually groups all entries where a certain field is equal. Each group can then be fed to an aggregation function.

Name	**Sex**	**Age**
John	Male	26
Mary	Female	22
Jake	Male	24
Ada	Female	30

Table 4.2: An example database table

Table 4.2 shows an example database, storing some information. Imagine you wanted to know the maximum age for each sex in your database. Assuming the database provides an SQL interface, a query similar to this could be performed:

```
SELECT Sex,MAX(Age) FROM Table GROUP BY Sex;
```

This would yield the result in table 4.3. Notice that a single row is returned for each unique value in the Sex column, and an appropriate average age is computed. [GMUW02]

Sex	**MAX(Age)**
Male	26
Female	30

Table 4.3: An example aggregation result

As can be seen, not only are the aggregation functions parallelizable by themselves, but they are also used on disjoint groups of entries in the database.

4.2.1 Simple distributed algorithms

Several algorithms for distributed database aggregation have been proposed, each with different strengths an weaknesses. We give a brief overview and comparison.

The repartitioning algorithm

Assume data is stored in a distributed database, with some data on each node. When doing an aggregation, the grouping attribute may have a value that is distributed across multiple nodes. The aggregation must include all entries with the same value in the grouping attribute. This is achieved by assigning each value to a specific node on which to run the aggregating function.

The repartitioning algorithm is implemented by transferring all data with a common grouping attribute to the same node, and doing aggregation there. Computing nodes are selected by running a hash function on the grouping attribute, and the grouping attribute and all fields necessary for the aggregating function is transferred to the computing node. As the node receives data, it updates the aggregate value for each entry it receives. When all the data has been transmitted, the results are output.

The two-phase algorithm

The repartitioning algorithm causes all data to be transmitted across the network, causing network bandwidth to become a likely performance bottleneck. [SM82] describes a two-phase algorithm where

intermediate results are computed on local nodes before they are transferred to a common node and merged. This can save significant amounts of data transfer, but only of if a significant amount of entries can be grouped on the node on which they are stored. Relating this to figure 4.4, the partial computation will be run on local nodes, while the merging step is computed after transferring the data to computing nodes.

A possible issue with the two-phase algorithm is that with data spread evenly across computing nodes, the first computing step may have to create intermediate values for every possible value of the grouping attribute. This introduces a significant risk of exhausting memory on the computing node. A common solution for centralised aggregation where the computation does not fit in memory is storing partial data in overflow files and read and process these files when memory is available [Bra03], but in a distributed environment other options also exist.

[Gra93] proposes a solution for this problem, which eliminates the need for storing intermediate values in persistent memory. When memory is exhausted, data is simply transferred prematurely to the node responsible of merging the result. With this scheme, several intermediate values may be transmitted for each value of the grouping attribute, causing increased network utilisation. However, one may argue that the considerably reduced complexity and reduced interaction with persistent storage make this extra traffic worthwhile.

Comparison of the repartition and two-phase algorithm

At first glance, the two-phase algorithm seems superior to repartition, due to the reduced network utilisation. However, this property relies on the possibility to combine a lot of values in the first step of the algorithm. [SN95] argue that this is not necessarily the case, since aggregation queries may also result in a large number of returned values, each being computed from few database entries. In such cases, computing intermediate values only creates overhead, by requiring additional computations.

For instance, given eight values spread evenly across four nodes, computing an aggregate value using the two-phase algorithm would require aggregating two values on each node in phase 1, plus merging four intermediate results in phase 2, a total of 12 operations. In comparison, the repartitioning algorithm would only require eight computations, but require a little extra network traffic.

Experiments performed by [SN95] indicate that the two-phase algorithm is well-suited for aggregations where many entries are combined, while the repartition algorithm shines where this is not the case.

4.2.2 Adaptive distributed algorithms

After pointing out that the two-phase and repartition algorithms have different strengths and weaknesses, [SN95] proposes three adaptive algorithms, designed to automatically select an appropriate aggregation scheme for any aggregation query at runtime, without prior knowledge about the number of groups created by the grouping statement. We introduce each algorithm briefly.

The sampling algorithm

Sampling is a technique which has been successfully used to estimate database parameters [Ses92]. Before doing the actual computation, each node picks some random entries and compare their grouping attributes. The number of unique entries gives a lower bound for the number of groups on the node, and can be used for deciding which algorithm to use in the actual computation. It can be shown that the number of samples needed on each node is fairly small, but that it is a function of the number of nodes, thus increasing as the number of computing nodes increase.

The adaptive two-phase algorithm

The adaptive two-phase algorithm starts using the who phase algorithm, assuming that data is normally aggregated into few groups. If the amount of groups happens to no longer fit in memory, the repartition algorithm is used instead. This approach is similar to the optimisation for the two pass algorithm mentioned in section 4.2.1, but differs by aborting the two pass algorithm completely, and instead completing the computation using the repartition algorithm. This is claimed to be a superior approach, because it eliminates the redundant local processing and memory utilisation.

The adaptive repartitioning algorithm

The adaptive repartitioning algorithm is the reversed version of the adaptive two-phase algorithm. It starts by assuming that there is a large enough amount of groups to justify using the repartitioning algorithm (usually based on hints from the optimiser), and initiates processing with this algorithm. Nodes doing aggregation investigates the first values they receive, and evaluates whether the amount of groups is large enough to justify using the repartitioning algorithm. If this is not the case, it notifies all other nodes that they should change to the adaptive two-phase algorithm.

Comparison of the adaptive algorithms

Because of the extra processing in the sampling phase, the sampling algorithm has an additional cost compared to the other algorithms. This cost is especially noticeable in systems with a large number of nodes, because the required size of the sample is a function of the number of nodes. In larger systems, this overhead exceeds that of the other two algorithms. The remaining algorithms appear to be quite equal, each being able to quickly adapt to the most efficient method. If a single algorithm was to be implemented, the adaptive two-phase algorithm seems to be the best choice.

4.2.3 Possible implementations in MapReduce

The data flows of the simple algorithms described in section 4.2.1 look suspiciously similar to the data flow in MapReduce. Having observed this, equivalent implementations in MapReduce follow immediately.

In the repartitioning algorithm, all entries in the *map* step are simply emitted immediately. In the two-phase algorithm, the *map* step calculates intermediate results for all groups, and emits these results. In both algorithms, the *reduce* step will simply combine any data it gets, using the desired aggregate function.

As can easily be seen, both functions can be efficiently implemented in the MapReduce library, at no apparent extra computing cost. Knowing this, we proceed investigating the adaptive algorithms introduced in section 4.2.2.

Provided that the MapReduce library supports fetching a few random items, the sampling algorithm may be implemented using two passes of MapReduce. One pass on a few random entries to determine what algorithm to use, and one for doing the actual computation.

The adaptive two-phase algorithm, being nothing but a combination of the two simple algorithms, is straight-forward to implement. The only feature required compared to naïve implementations of the simple algorithms is a possibility to detect that memory is exhausted, and this is most likely possible. The adaptive repartitioning algorithm, however, requires inter-node communication. Such communication is not provided by MapReduce, so this approach could not be implemented.

Summarising, both the repartitioning, two-phase and adaptive two-phase algorithms can be easily implemented in MapReduce at no extra cost. The sampling algorithm might be possible to implement

depending on the feature set of the MapReduce library, and the adaptive repartitioning is impossible to implement. Knowing that the adaptive repartitioning algorithm normally has similar performance to the adaptive two-phase algorithm, and that the sampling algorithm is normally slower, we conclude that the MapReduce library is well suited for doing database-like aggregation.

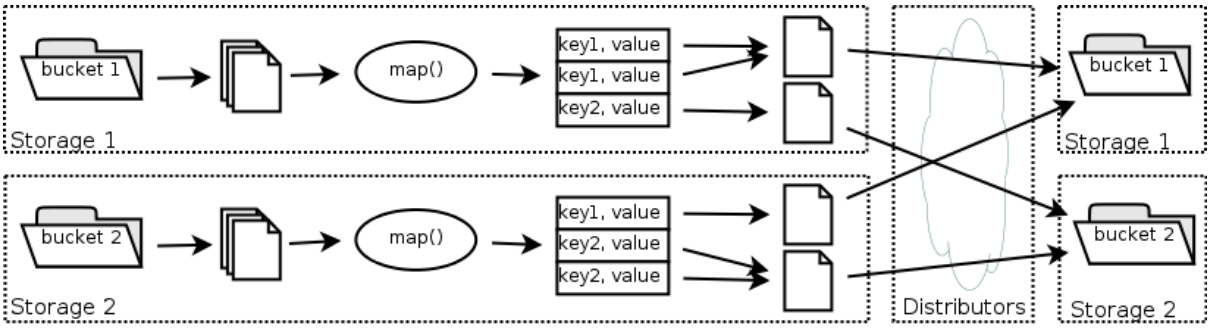

Figure 4.5: Overview of the *map* process in the VDS implementation of MapReduce

4.3 MapReduce in VDS

At an early stage of the work for this thesis, a MapReduce library for VDS was developed. The implementation is based on the visitor design of both *map* and *reduce* in [Gry06], and all output is stored back into VDS to allow further processing or retrieval.

Both the *map* and *reduce* implementations rely on specialised server-side visitors doing preprocessing. Unlike other VDS visitors, the MapReduce visitors do not return any data to the client. Instead, output data is fed back into VDS using multiple Put requests initiated by the visitor. This design allows redistribution of data without relying of client bandwidth.

As described in section 3.3.2, entire buckets are handled by the same visitor instance, which handles a configurable amount of one or more buckets at a time. This has the effect of splitting the dataset into groups of one of more buckets, comparable to the M and R parts used in Google's MapReduce implementation (see Section 4.1.3).

4.3.1 The *map* step

The *map* step in VDS differs from the Google equivalent in the type of input data. Where the Google implementation expects tuples, the VDS variant accepts documents, which may be parsed as desired by the map function.

4.3.2 The *reduce* step

The *reduce* step behaves almost like map, except for the input format. The map function is, like in Google's implementation, called with a key and a list of values. The key and values has previously been collected from all relevant documents in the bucket.

4.3.3 Regrouping of data

Output data from the *map* step is placed in separate groups using a hash function on the key part of each tuple. The data in each group is saved in one or more documents, depending on the size of the data. To allow all *map* output to be grouped correctly for the *reduce* run, the `userdoc` scheme (see Section 3.5.1) is used to name an explicit bucket for each group. The buckets are usually selected from a (configurable) subset of the total amount of buckets, since *reduce* is usually performed with considerably less data groups than *map*. See Figure 4.5 for an illustration of this process.

This usage of documents to regroup the data requires a minimum of $n * m$ documents to be stored into VDS, where n is the total amount of buckets data is being processed from, and m is the amount of buckets the data is being stored to. If some data is too large to fit in a single document, it must instead be split into several documents.

To guarantee that all documents used for saving *map* output is unique, the document ID of each output document consists of the following values:

- A unique ID identifying the MapReduce run being performed
- The source bucket number for the computation
- The destination bucket number
- A counter, for use when data must be split in multiple documents
- A string stating the type of output (either "map" or "reduce"), allowing the same format to be used for both *map* and *reduce* output

The document ID format described above not only guarantees that all parts of the output is assigned a unique ID, but is also completely deterministic. This is important when parts of the computation fails and has to be repeated, since any partial data previously being saved is overwritten, efficiently avoiding duplicate data.

4.3.4 Memory requirements

With the current implementation, all tuples created from a bucket is kept in memory while the bucket is being processed. This data is assumed to be significantly smaller than the bucket itself, and this is therefore not believed to cause any unsolvable problems. However, this implies that there is a limit to how many buckets one can process in parallel on the same node.

Assuming documents evenly distributed between buckets, the memory requirement $M_{MapReduce}$ for processing a single bucket is:

$$M_{MapReduce} = \frac{|D|}{|B|} * s$$

where D is the set of documents, B is the set of buckets, and s is the combined size of all tuples being returned from processing a single document.

Chapter 5

MapReduce-related VDS configuration

This chapter discusses what VDS operations are performance-critical for a MapReduce application built on top of VDS. It then briefly describes the experiment framework and hardware, before focusing on each relevant configuration item in sequence. For each configuration item, suitable settings are determined by the help of theoretical and experimental approaches.

5.1 VDS features relevant to MapReduce

The MapReduce implementation used in VDS makes extensive use of the Visiting feature in VDS, which is used for both the *map* and *reduce* steps. In addition, the Put operation is used for storing *map* output back into VDS. These two features are therefore ideal candidates for performance tuning from a MapReduce point of view.

5.1.1 Visiting

The visiting feature of VDS is a central part of the MapReduce implementation. However, not all aspects of visiting is relevant. Since input data for both *map* and *reduce* is processed directly on the storage nodes and never transmitted over the network, the only network traffic generated by the visitor are administrative messages. This makes network bandwidth an unlikely bottleneck, and leaves the following possible limitations on MapReduce-related visiting performance:

- Network latency on administrative messages
- Processing power on computing storage nodes
- Available memory on storage nodes

5.1.2 Put

Both the *map* and *reduce* process output data by storing it in VDS. This is done using multiple Put requests directly from the visiting process. The amount of Put requests depend on the amount of buckets with input data and the amount of output buckets (see section 4.3). Additionally, output larger than a certain size might be split into multiple smaller documents, further increasing the amount of puts. The following factors may limit Put performance:

- Bandwidth limitations
- The amount of documents
- The size of data sent to a specific bucket

5.2 Experiment methodology

All experiments are performed three or more times to ensure reproducible results, and average values from all successful test runs are used. We give a brief description of the hardware and framework used.

5.2.1 Experiment hardware

Two clusters have been used for the experiments in this thesis. One consisting of ten nodes, the other of two. We briefly list the most important hardware characteristics of both:

The ten-node cluster

- Dual Xeon 2.8GHz processors
- 2 GB RAM
- Five 73GB 15000 RPM SCSI320 hard drives
- FreeBSD 4.10 (eight nodes) or 4.11 (two nodes)

The two-node cluster

- Dual Xeon 3.0 GHz processors
- 4 GB RAM
- Two 250GB SATA hard drives
- FreeBSD 4.11

5.2.2 Experiment framework

To allow identification of the limiting resources of operations, monitoring of CPU, memory, network and I/O utilisation is a great aid. Such a solution is Munin [Mun], an open-source monitoring framework developed by the Norwegian company Linpro [Lin]. Munin provides a rich set of plugins out of the box, and monitoring of most desirable aspects were already supported.

Unfortunately, Munin is locked to a 300 second update interval, providing an insufficient granularity for this use. Also, Munin operates by installing a server on each node and requires polling over the network. Furthermore, separation of data on a per-experiment basis proved cumbersome, as Munin is designed for continuous graphing and not aggregation. To take advantage of Munin's readily available plugins, a simple wrapper was created. The wrapper is implemented as a daemon appending data to simple text files every fifth second.

After performing experiments, monitoring data was gathered from all participating nodes and used for graphing with RRDTool and other aggregation. By deferring this transfer and aggregation until after the completion of experiments, the monitoring overhead was limited to a simple daemon and a few small file appends, both at an assumed negligible cost.

5.3 Tuning the maximum slotfile size

By adjusting the maximum slotfile size, a VDS administrator can control the amount and size of slotfiles in buckets. This setting may be changed on a running VDS cluster, and will take effect on slotfiles as soon as data is written to them. Slotfiles with no write requests will remain unchanged, unless another slotfile in the same bucket triggers a slotfile split.

Storage nodes are the only part of VDS with a concept of slotfiles, and the size of slotfiles will thus only affect storage nodes. Much like the amount of buckets, the size (and thus indirectly amount) of slotfiles should both affect visiting latency and memory spent when reading or writing them.

During a Put operation, which only writes to a single slotfile, a small slotfile should imply that less data must be read and written, and this would at first glance mean that slotfiles should be as small as possible. However, a too small maximum size would cause slotfiles to be split frequently, and this is likely to affect performance.

During visiting, a large amount of files would imply frequent opening and closing of files, both of which have a non-zero cost. The amount of seeks would also increase, as each open implies at least one random read. This is likely to provide inferior performance compared to a streaming read of a large slotfile.

An suitable default value for both Put and Visit operations is likely to exist, and this value quite possibly depends on the amount and size of documents to be stored in the slotfile.

5.3.1 Effect of slotfile size on Put performance

Because MapReduce is implemented in a manner possibly causing a lot of data to be stored in a limited amount of buckets (see Section 4.3), a MapReduce program may end up saving a lot more data in buckets than what is done during normal VDS operation. Therefore, prior to running a MapReduce application, allowing maximum data to be stored in each bucket may be in order. As discussed in Section 5.3, the slotfile size stands out as the most likely option to be effective for such tuning. If the size limits of slotfiles are changed, these changes take effect the next time a Put request affects the slotfile.

Very large file sizes and a great amount of documents per slotfile is expected to be limiting factors of Put operations, and to assess both of these the experiments are split in two: One for rather small documents in large amounts, and one for very large, but fewer documents.

Experiments were performed with the two-node cluster described in Section 5.2.1. To allow measuring of hardware utilisation related to storage node activity only, one node is configured as a dedicated storage node, while the other is used as both distributor and client. The experiment storage node has been configured with only 10 buckets. The selected value of 10 buckets is far below the recommended amount for such a dataset, and the system is therefore expected to struggle. By observing such a struggling node we hope to also determine what factors limit the amount of data to be stored in a single bucket.

In these experiments, one Put request per document is issued in a sequential manner. However, the requests are issues asynchronously, meaning multiple requests are running throughout the experiments. The Put requests are queued at both distributors as well as storage nodes before being allowed to proceed through the system.

Results with small documents

A plot of some metrics collected from putting 300.000 small documents with various values for the maximum slotfile size is shown in Figure 5.1. As can be seen, all included slotfile sizes are quite small, but the time spent putting documents is already rising steadily, indicating that further increasing the value is of little use. We will discuss the plausible causes of this behaviour below.

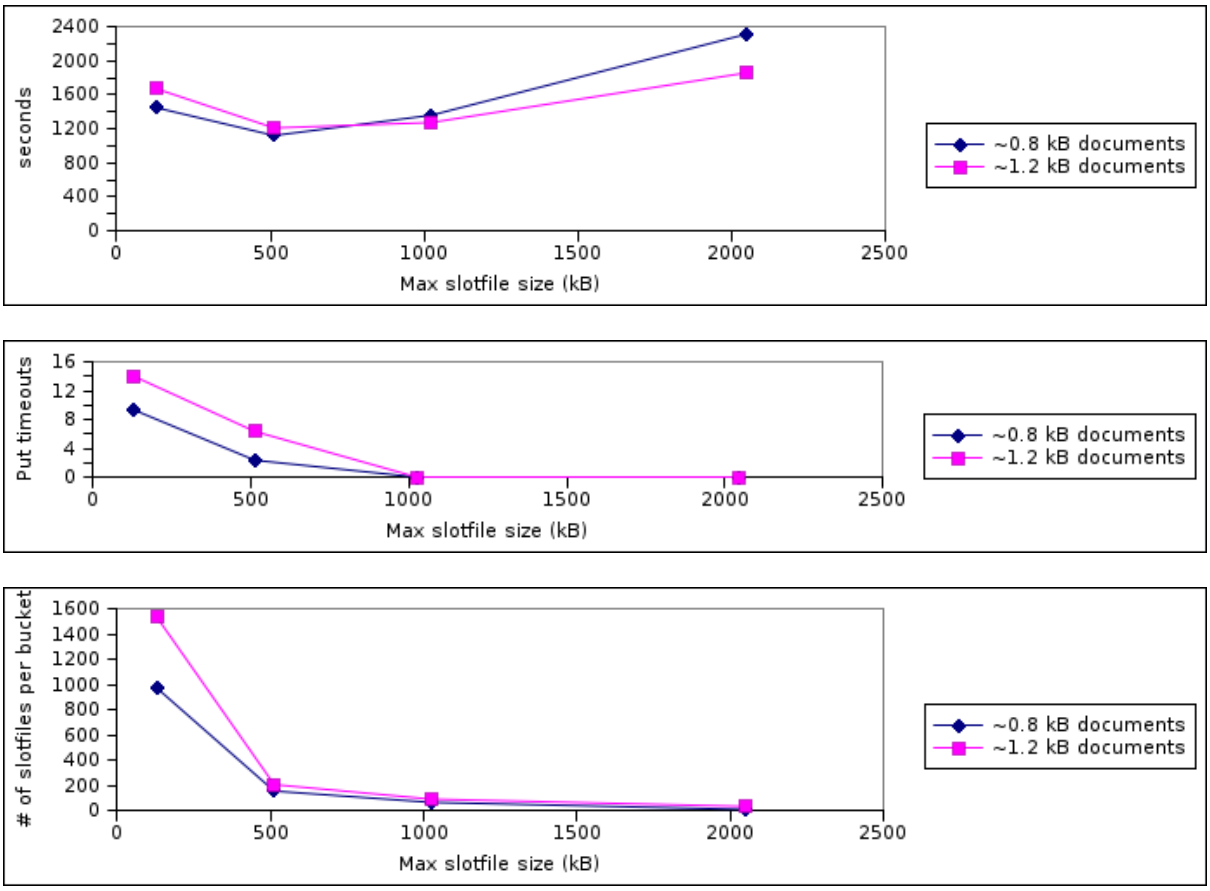

Figure 5.1: Time spent, number of Put timeouts and number of resulting slotfiles when putting 300.000 small documents with differing slotfile sizes

During testing with very small slotfiles, some Put request timeouts occurred. Comparing the number of timeouts and the amount of slotfiles in Figure 5.1 hints that this behaviour might be related to the amount or size of slotfiles. More specifically, slotfile splitting is a likely culprit. As described in Section 3.5.2, all slotfiles in a bucket are split at the same time, causing this operation to be very expensive as the amount of slotfiles increase. According to these results, splitting 64 slotfiles at the same time appears to cause Put requests to time out.

To aid in identifying the factor limiting performance when doing Put requests, measuring of CPU utilisation and network traffic has been performed (see Figure 5.2). Similar measuring of memory utilisation was also performed, but all showed the same: Practically no memory is active, and the remaining is used for operating system buffers and cache.

As can be seen in the bandwidth charts of Figure 5.2, the smaller slotfile sizes appears to give superior throughput in the beginning of the tests. During the first five minutes of the 128 kB slotfile test, we can clearly see that throughput is higher than in the other tests, but quickly decays. A similar effect can be seen in the 512 kB test, however to a lesser extent and at a later time. This is consistent with the suspicion that frequent slotfile splits cause reduced performance, since increasing amounts of slotfile splits are performed while the amount of data in the bucket increases. This indicates that Put is bound to the performance of slotfile splitting when buckets are split over very many small files.

Comparing the charts further, the shape of the CPU utilisation chart closely resembles the outgoing network traffic. As described in Section 3.6.1, the outgoing traffic is mainly replies and acknowledgements,

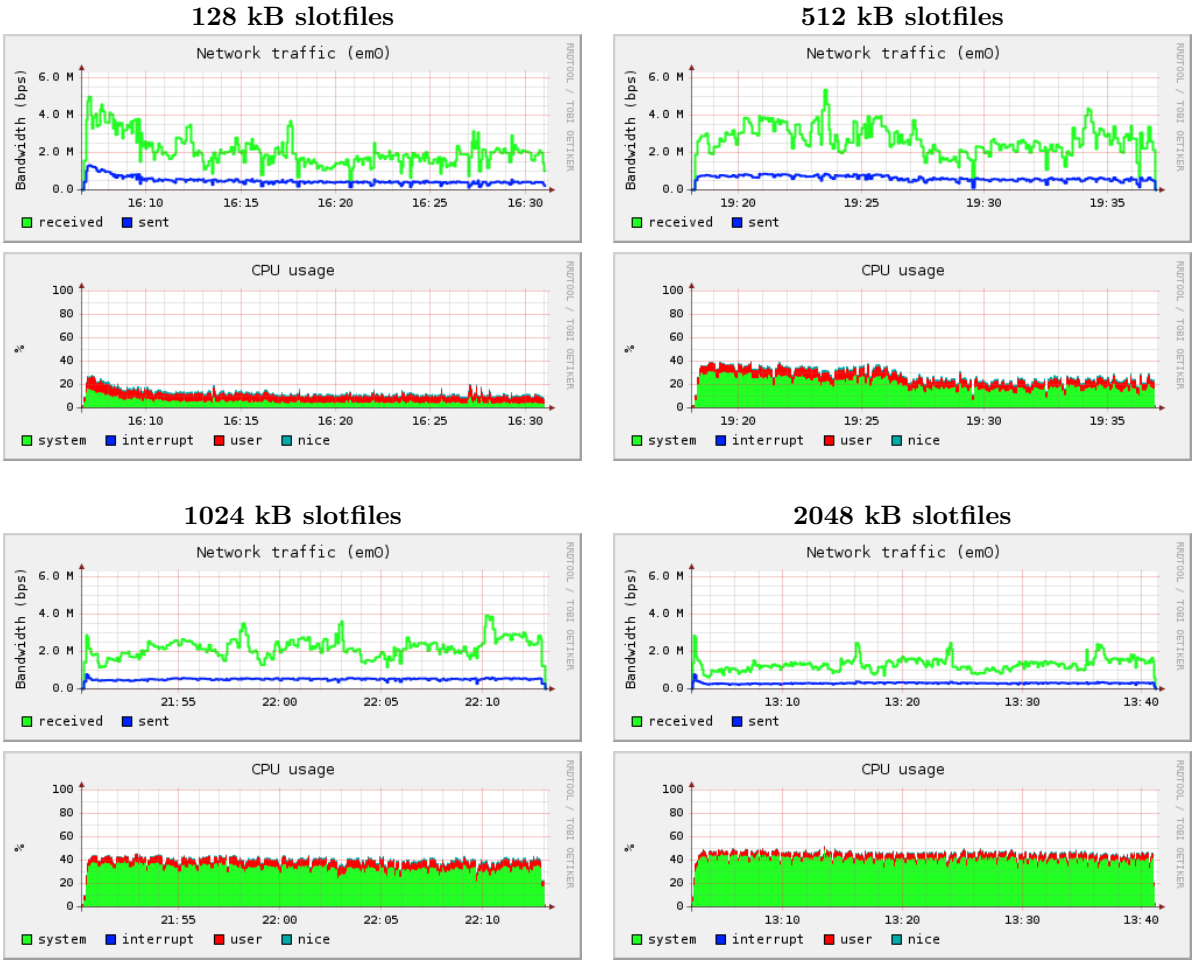

Figure 5.2: CPU and network utilisation while putting 300.000 documents of average size 0.8kB

and this value should thus indicate the amount of documents currently being processed. While the 128 and 512 kB CPU utilisation charts decrease over time, the 1024 and 2048 kB charts show a steady 50% utilisation. Since the experiment was executed on dual CPU nodes, this amounts to one fully utilised CPU. Considering that the storage node part of Put is single-threaded, and that the utilisation does not change as slotfile sizes are further increased, this could indicate that Put is CPU-bound when the metadata list grows very large. This list has to be searched once per Put request, as described in Section 3.6.1, and this is a CPU-intensive operation.

Results with large documents

As discussed above, Put appears to be CPU-bound when inserting very many documents into the same slotfile. We will now attempt to cluster data in fewer and larger documents, to see how this affects performance under differing slotfile sizes. Figure 5.3 shows graphs showing various characteristics collected from test runs doing Put of 20.000 documents of size 155 and 215 kB.

Comparing Figure 5.1 and 5.3, we notice that the shape of the time graphs differ. Unlike the small document experiment, the time spent putting large documents does not seem to rise significantly when increasing the slotfile size. Admittedly, the value for 155 kB large documents with a 16MB slotfile size is somewhat smaller than the larger slotfile sizes, but since the 215 kB documents do not share this

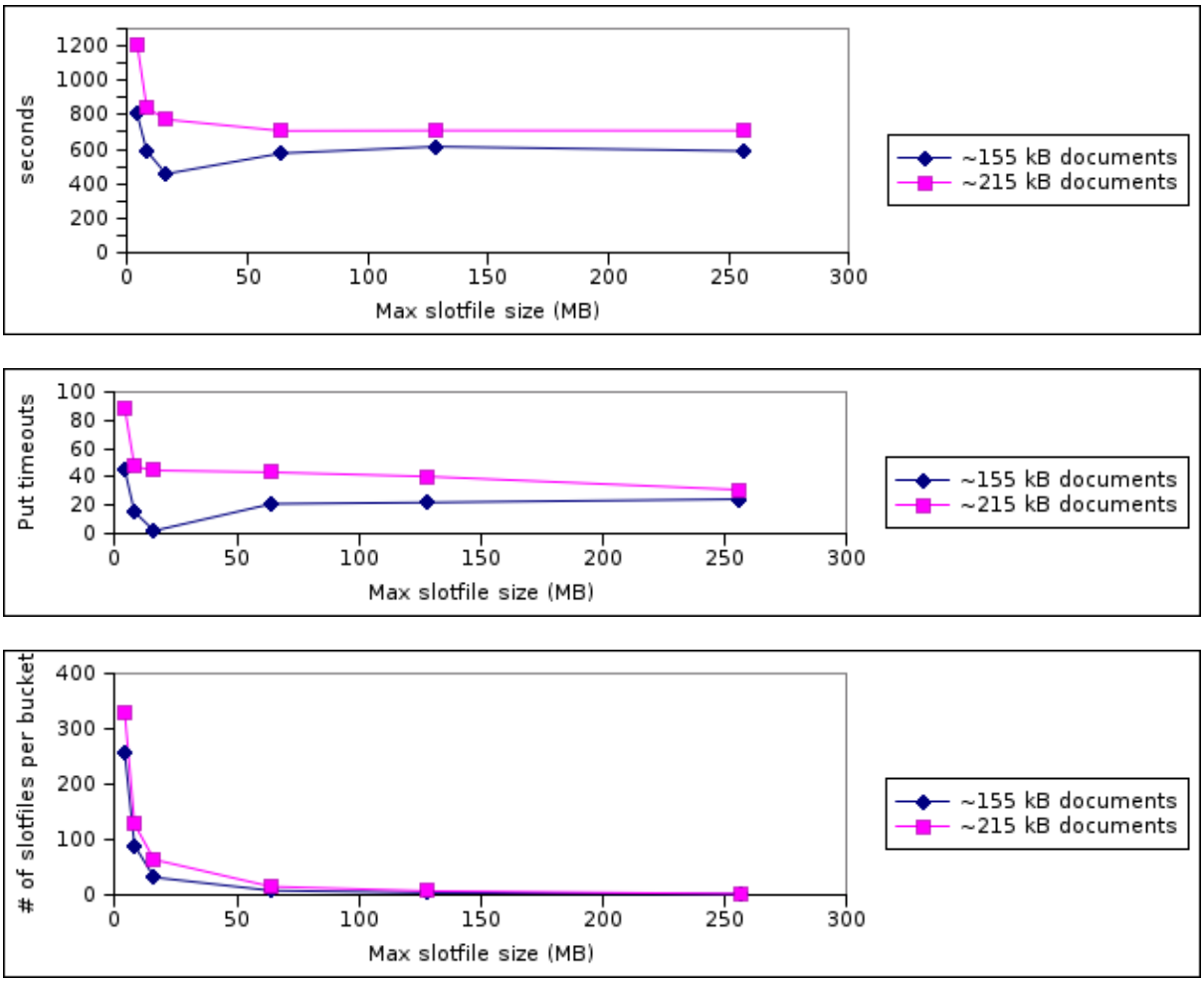

Figure 5.3: Time spent, number of Put timeouts and number of slotfiles when putting 20.000 large documents with differing slotfile sizes

property, the difference is not likely to be coincidental and not general among all data sets. It did however occur in all test runs with the same dataset and settings.

Figure 5.3 also shows numerous Put timeouts, and unlike the experiment with small documents, the timeouts do not seem to stop when the slotfile size is increased and the slotfile splitting is reduced. Instead, the graphs showing total time spent putting and amounts of timeouts look strikingly similar. Because of this similarity, we believe these timeouts are simply caused by the node being slightly overloaded throughout the experiment, causing some documents to time out now and then.

Looking at data collected from the storage node, we observe that all test runs show a CPU utilisation of about 30% or less during the majority of the tests, and only rarely passing 40% for a few seconds. They do show a very slight increase in CPU utilisation as the slotfile sizes are increased, but never cause the operation to be CPU bound. This supports our hypothesis of Put being CPU-bound due to the amount of documents, and not the combined size.

As previously mentioned, the small document test showed virtually no active memory. This is not the case when putting large documents. Instead, the active memory seems to be about twice the maximum slotfile size, as Figure 5.4 quite clearly shows. Considering that the small document test was performed with significantly smaller slotfile sizes, these results do not contradict each other.

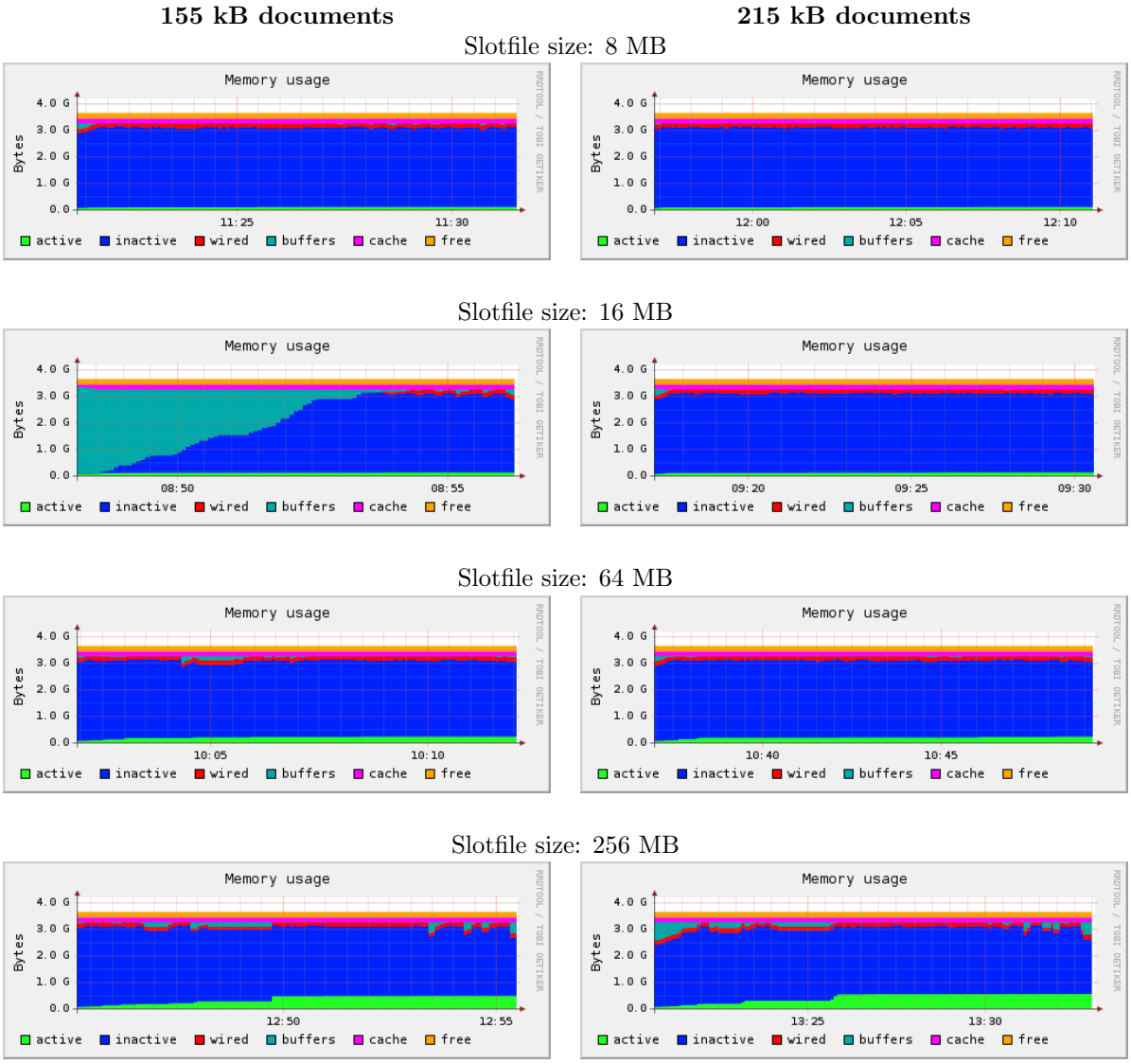

Figure 5.4: Memory utilisation while putting 20.000 documents of average sizes 155 and 215 kB

We also notice that the experiment with 155 kB documents and 16 MB slotfiles appears to show far less inactive memory. Inactive memory largely consists of cached data from previous disk reads. We see from Figure 5.3 that this exact combination also both reduced the amount of timeouts and the total time spent putting documents. A possible explanation for this could be an increased amount of cache hits using this particular combination, causing fewer reads and thus less cached data. If this is the case, the effect is probably coincidental, and not likely to be exploitable in optimisations.

Regarding throughput, plotting network traffic and disk throughput next to each other shows a striking similarity, as shown in Figure 5.5. As can quite clearly be seen, network and disk throughput are near opposites. A plausible explanation for this behaviour is that network traffic is reduced when a lot of data is synced to disk. In other words, the operation is I/O-bound. This is not a surprising result, with the experiment being performed on a single-disk node. Also worth noting is that disk throughput never exceeds 10 MB/s. An attempt to copy a few GB of large files show a maximum throughput of about 30MB/s, by far exceeding the throughput in VDS. A likely cause for this is how VDS accesses several

Figure 5.5: Network and disk utilisation while putting 20.000 documents of average size 215 kB

locations in each slotfile, which is considerably slower than a long, streaming write [Sta01]. If this is the case, further improved buffering and merging of I/O operations might improve performance.

5.3.2 Effect of slotfile sizes on Visit performance

After testing Put performance with varying slotfile sizes, a similar experiment was performed for Visit, to determine which slotfile sizes are suitable for Visiting of buckets with a lot of data. This experiment could be used to determine whether the same suitable slotfile sizes exist for both Put and Visiting, and if not be used to determine a suitable compromise.

Experiment setup

This test was performed with the same data and hardware as the Put counterpart. VDS was configured with the default two threads per disk, meaning a total of two threads. All other settings were also left at their default value.

To perform this experiment, a special no-op MapReduce application was made, designed to simply discard all data from documents as they were read from disk. As mentioned in 5.1.1, the MapReduce visitor

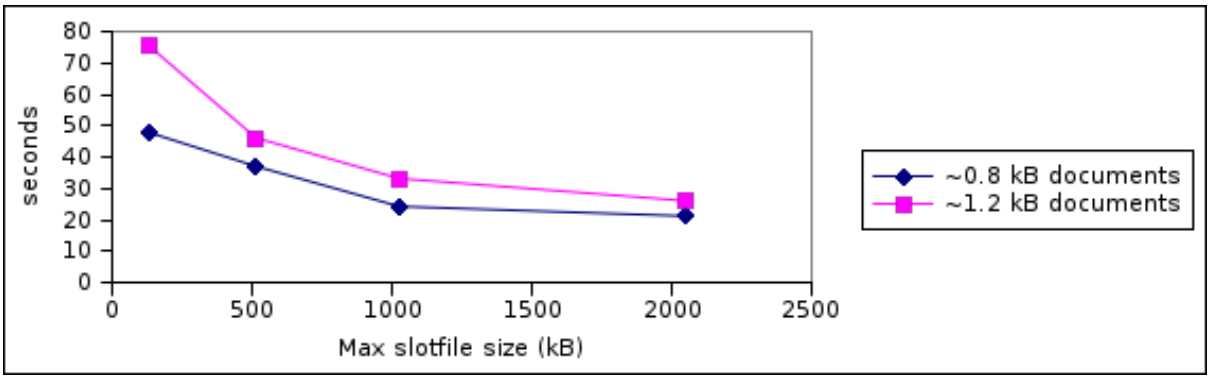

Figure 5.6: Time spent when visiting 300.000 small documents with differing slotfile sizes

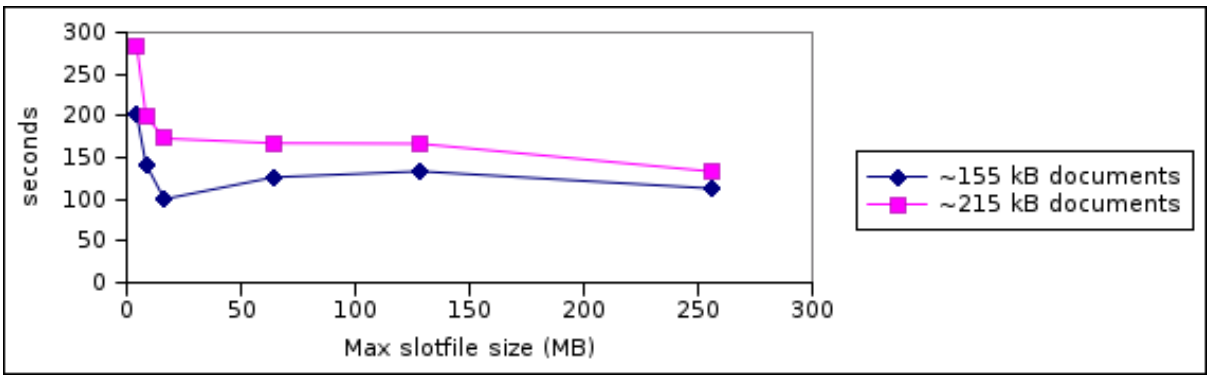

Figure 5.7: Time spent when visiting 20.000 large documents with differing slotfile sizes

does not return data the normal way, but instead uses a series of Put requests to store values back into VDS. This was done by implementing *map* as a function with an empty body, and omitting the *reduce* step completely when running the application. With this setup, the application will simply measure disk throughput, ensuring that network traffic and computing does not complicate the experiment. Complete source code for the no-op MapReduce visitor is included in Appendix A.

Results

Studying Figure 5.6 and 5.7, we notice that they do indeed closely resemble their Put counterparts in figure 5.1 and 5.3. There is however one exception, and that is the case of very many small documents in the same slotfile. When doing Put, the time spent is rising steadily, while the Visit counterpart is simply decreasing. Again, this indicates that searching the metadata list is the limiting factor when doing Put, since Visit does not perform this search for each document.

All tests show less than 20% total CPU utilisation, indicating that CPU it not a limiting factor. The amount of active memory is identical to the Put counterpart, but no memory is spared for OS-specific caches, indicating that VDS uses all available memory for its own internal cache.

While visiting both small and large documents, disk throughput was measured. Unfortunately, the measuring granularity of 5 seconds caused the values for small documents to have a too low resolution to be considered. Graphs of I/O throughput when visiting 20.000 documents of average size 155 kB are provided in Figure 5.8. The 215 kB graph shows near identical results, and has therefore been omitted.

As can be seen, throughput is pretty even at about 10 MB per second using 155 kB documents, regardless

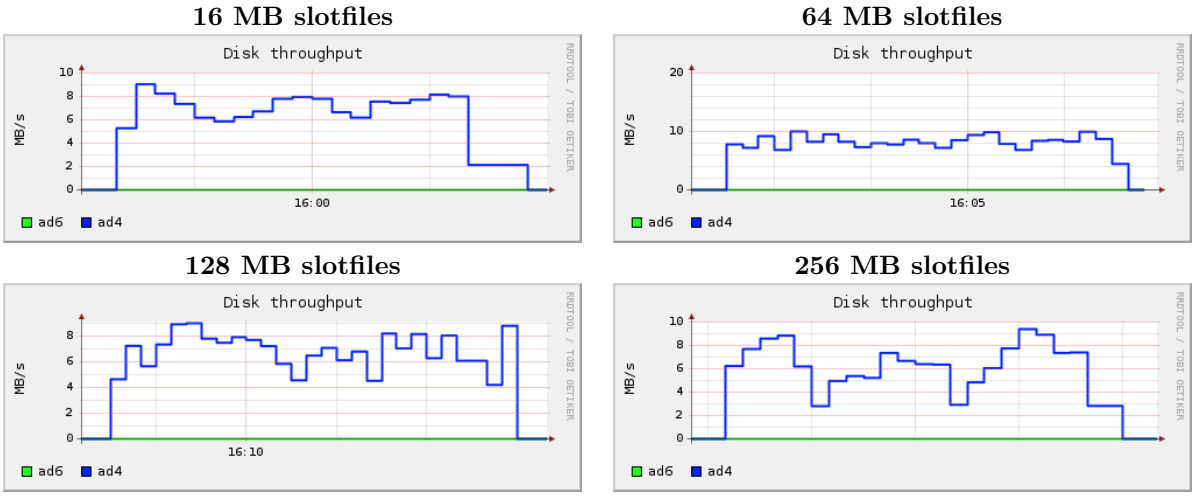

Figure 5.8: Disk throughput while putting 20.000 documents of average sizes 155 kB

of slotfile size. This is one third of what the disk is capable of when reading from the underlying file system in large, streaming reads. This is probably due to visiting doing smaller reads, but the time spent adding documents to docblocks and related tasks may also contribute. We also notice a few dips in the graph, growing larger and rarer as the slotfile size is increased. The cause of these dips are unclear, but they may be related to the amount of slotfiles. As shown earlier, They do not seem to affect total throughput much, an we will therefore not investigate them further.

5.3.3 Suggested settings

The results of Section 5.3.1 and 5.3.2 indicate that for optimum Put and Visit performance, the amount of slotfiles should not exceed 64. When considering Put performance, slotfiles can be arbitrarily large in file size, but not exceed a certain amount of documents per slotfile. Also, the size of the slotfiles have an impact on the memory requirements of both Put and Visit. We discuss the optimal settings of maximum file size and amount of documents separately.

The maximum file size

When selecting the maximum file size, one must consider how much memory that can be dedicated to a single put or visiting process. Section 6.1.1 will also find that when putting more than 64 MB into the same bucket, performance issues ensue regardless of slotfile and document sizes. Based on this, the existing VDS default of 16 MB seems feasible, and we do not attempt any automated adjustment of this default.

The maximum amount of documents per slotfile

When considering the maximum amount of documents per slotfile, this is a somewhat different story. Because nodes may have differing computational power, simply setting a fixed value might not be suitable for all hardware. However, since an excessive amount of documents in a single slotfile only affects the Put operation, this operation can be used for automated tuning. We propose an algorithm for this purpose:

First, initialise the maximum number of documents to infinity. As the system is populated with documents, the number of documents in slotfiles will increase. As soon as a node detects that the current Put request is CPU-bound (either by monitoring load directly or checking the time spent searching through the metadata list), it messages the config server stating that the amount of documents currently in the slotfile is an upper bound. Eventually, the slowest node will have reported its maximum number of documents, and the setting will converge.

However, this algorithm does not solve the problem completely. There remains the issue of setting the threshold value. In a system where data is primarily read-only, CPU-bound Put requests may be perfectly acceptable, as long as visiting time is kept limited. In this case, an elevated reporting threshold may be used.

Also, in heterogeneous systems, one might not wish to allow a single node to limit the slotfiles on all other nodes, but rather let Put stay CPU-bound on this particular node[1]. In this case the config server may require multiple nodes to report a lower value before lowering the global setting. Put on the affected nodes will stay CPU-bound until the setting is changed, but will cease being so immediately after, as the next Put triggers a slotfile split.

In a dynamically changing cluster, the slowest nodes may some day be replaced. To allow the improved performance to take effect on the maximum slotfile size when the slowest nodes are replaced, the setting may simply be reset to infinity and the algorithm will find a new value. This may also be used after adjusting the threshold for reporting the maximum document count, for instance after collecting statistics on usage patterns of the particular system.

[1] VDS might solve this problem by implementing more sophisticated load balancing in the future.

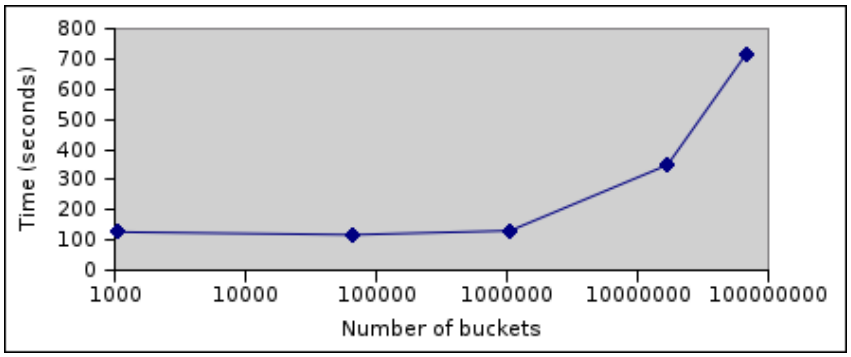

Figure 5.9: Time spent visiting various amounts of empty buckets (note that the slotfile count axis is logarithmic)

5.4 Tuning the number of buckets

The number of buckets in a system has a possible impact on both visiting and overall performance. During visiting the opening and closing of each bucket represents a constant cost, and this indicates that fewer buckets might improve visiting latency. However, with larger buckets more data must be kept in memory at once, and this is likely to cancel out the memory saved by reducing the size of the system map.

Since a Put operation only accesses a single bucket, the number of buckets should not affect the Put operation. Larger buckets are a likely consequence of a reduced amount of buckets, but Put only accesses a single slotfile in a bucket, and the size of slotfiles is handled by other configuration options.

The increased memory requirements of large buckets combined with the increased visiting latency of a large amount of buckets indicate that an optimal value exists, and that this value depends in the usage patterns of the system.

We attempt to verify this statement by visiting various empty VDS installations. Since all buckets are empty, the storage node visitors will complete immediately, leaving only the overhead of the visiting behind.

5.4.1 Results

Figure 5.9 shows the time spent visiting a VDS system with various amounts of buckets. As can be seen, the time spent does indeed increase noticeably when the amount of buckets is increased. However, note that when using 67.108.864 (2^{26}) buckets, the total time spent is still only about 700 seconds. Such a small number would most likely drown in the time required for other work when visiting a node with data on it.

Investigating data collected on the nodes, we notice a 100% utilisation of one of the processors on the client. The client being single-threaded, this effectively means that the visiting is CPU-bound on the client. Note however that this is when visiting a completely empty system, and that other time-consuming work when doing a regular visit will most likely be a more limiting factor.

5.4.2 Suggested settings

The results from Section 5.3.1 indicate that if the amount of slotfiles in a single bucket exceeds 64, there is a considerable performance penalty. This defines the minimum feasible amount of buckets in terms of

the number of slotfiles.

Considering the results of Section 5.4.1, the amount of buckets may be set very large in a VDS system without considerable performance impact from the amount of documents by themselves. However, the bucket count also forms a lower bound on the total amount of slotfiles in a system, so setting an arbitrarily large value will most likely cause reduced performance, similar to the effect discovered in Section 5.3.2. The amount of buckets should thus be plentiful, but not completely exaggerated.

When considering MapReduce, the total amount of buckets in the system also affects the amount of documents created by a MapReduce run across all buckets. This means that a system with very many buckets will cause a lot of documents to be saved into the same bucket by the MapReduce visitor.

The minimum amount of buckets

Assuming documents spread uniformly across all buckets, the minimum amount of buckets may easily be calculated. Given a document collection D with total size s_D, maximum amount of documents per slotfile *slotfilecount*, maximum size of slotfiles *slotfilesize* and maximum acceptable amount of slotfiles per bucket *maxslotfiles*, the minimum amount of buckets is:

$$\max\left(\left\lceil \frac{|D|}{slotfilecount * maxslotfiles} \right\rceil, \left\lceil \frac{s_D}{slotfilesize * maxslotfiles} \right\rceil\right)$$

As can be seen, this assumes that the amount of documents, size of documents and maximum amount of documents for each slotfile is known. Since there is no way VDS can detect the planned amount and size of documents to be stored in it, these values must be estimated manually. If the algorithm proposed in Section 5.3.3 is used to determine the maximum amount of documents in slotfiles, the final value of this setting is also unknown. However, this may be estimated by setting up a smaller system with few buckets, setting a very large maximum slotfile size, and feeding it with a random sample of data. The value produced by the above algorithm may then be used to estimate the desired amount of buckets.

Acknowledging that the data is unlikely to be completely uniformly spread across buckets, the value output by this method should not be used as is, but used as a guideline. Most likely, the result may be multiplied with some factor to compensate for unevenly spread data, but this factor will depend on the nature of the data.

Maximum amount of buckets for allowing successful MapReduce runs

As described in Section 4.3, the MapReduce implementation requires a minimum of $n * m$ documents, where n is the amount of buckets data is collected from, and m is the amount of buckets data is written to. In the minimal case, where no documents have to be split, each bucket must contain one document for each source bucket. This gives a minimum amount of n documents in each bucket.

From this simple observation, we learn that the maximum amount of buckets in the system equals the maximum amount of documents in a single bucket, if MapReduce operations are to be completed successfully with the current implementation. This equals the maximum amount of documents per slotfile multiplied with the maximum acceptable amount of slotfiles in a single bucket. The maximum amount of documents per slotfile may either be set explicitly or calculated using the above algorithm, while the maximum amount of slotfiles in a single bucket was found to be 64 in Section 5.3.1 and 5.3.2.

As a side note, the amount of puts done by the MapReduce implementation is a function of the amount of buckets used for input data (see section 4.3), so decreasing the amount of buckets may also reduce the amount of Put commands required. By comparing the bandwidth graphs of Figure 5.2 and 5.5, we see that larger documents may improve I/O throughput. However, since the MapReduce framework is targeted against aggregating data in already running systems, where the number of buckets may not

be changed, this optimisation is rarely an option. We will therefore not investigate this matter further. Also, there is likely to be a limit to how large documents may be. When this limit is reached reducing the number of buckets will only require output documents to be split, completely cancelling the effect. These limits will be further examined in Section 6.1.

5.5 Pre-calculating of the number of slotfiles per bucket

Section 5.3 shows that the number of slotfiles per bucket has an effect on the performance of both Put and Visit. Thus, knowledge of the relation between configuration options and the expected number of slotfiles per bucket would be a useful aid when configuring VDS. We attempt to create such an estimate, and later verify its correctness.

Assuming all documents are spread evenly across all buckets, the average total size of data in each bucket will be

$$\bar{b} = \frac{\bar{d} * |D|}{|B|}$$

where $\bar{b}$ is the average size of buckets, $\bar{d}$ is the average size of documents, D is the set of documents and B is the set of buckets.

Provided the data is spread evenly across all slotfiles in each bucket, the number of slotfiles is the nearest power of two large enough to have room for all data. Given the maximum slotfile size *slotfilesize*, the above data and some integer x, the maximum number of slotfiles $|S_{bucket}|$ in a bucket is given by:

$$|S_{bucket}| = 2^x \geq \frac{\bar{b}}{slotfilesize} > 2^{x-1}$$

Solving for the lowest integer value of x, we get:

$$\begin{aligned} 2^x &\geq \frac{\bar{b}}{slotfilesize} \\ x &\geq \log_2\left(\frac{\bar{b}}{slotfilesize}\right) \\ x &= \left\lceil \log_2\left(\frac{\bar{b}}{slotfilesize}\right)\right\rceil \end{aligned}$$

Which gives us an estimate for $|S_{bucket}|$:

$$\begin{aligned} |S_{bucket}| &= 2^{\left\lceil \log_2\left(\frac{\bar{b}}{slotfilesize}\right)\right\rceil} \\ |S_{bucket}| &= 2^{\left\lceil \log_2\left(\frac{\bar{d}*|D|}{|B|*slotfilesize}\right)\right\rceil} \end{aligned}$$

Note that this estimate assumes a perfect distribution of documents, both across buckets and slotfiles. If data is not uniformly distributed between slotfiles inside buckets, additional splits will occur. Since a perfect distribution is unlikely, adding a margin of error is in order.

We are more interested in the maximum value than an average, because the performance penalty occurs as soon as a single bucket contains too many slotfiles. Therefore, slight overestimation is better than underestimation.

A single additional slotfile split is the smallest possible margin of error providing an amount of slotfiles that may occur in practice, and gives twice the amount of slotfiles. We will therefore multiply our estimate with 2 in our evaluation.

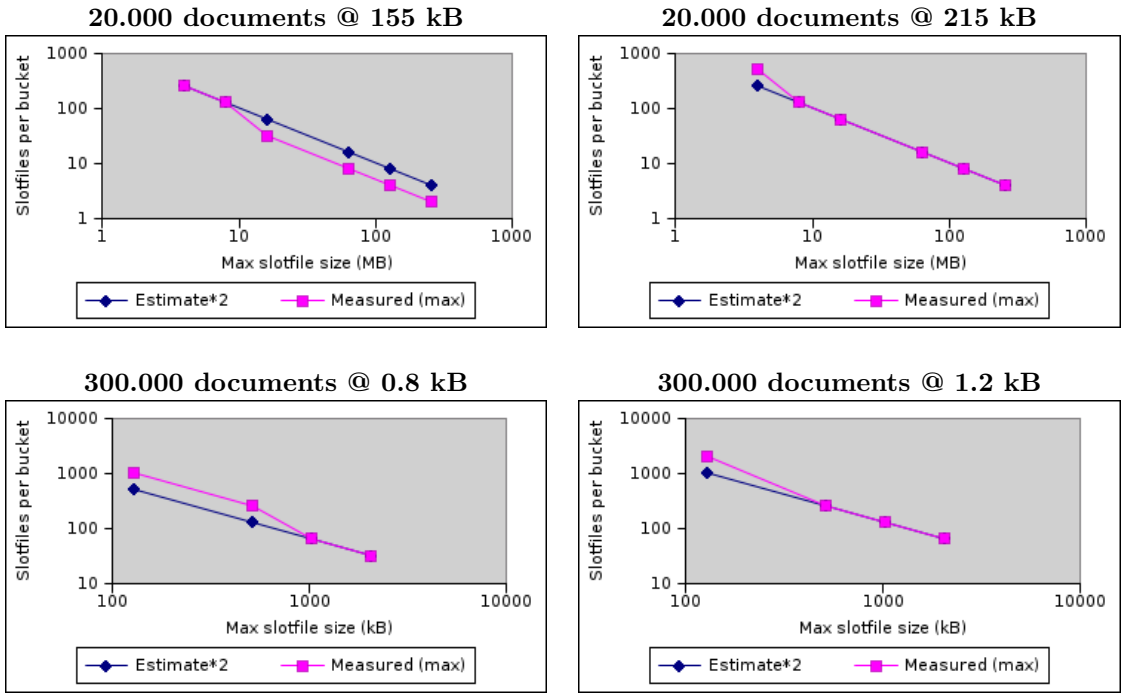

Figure 5.10: Estimate of number of slotfiles compared to measured values using 10 buckets

20.000 documents @ 155 kB

File size	Estimate*2	Measured (max)
4 MB	256	256
8 MB	128	128
16 MB	64	32
64 MB	16	8
128 MB	8	4
256 MB	4	2

20.000 documents @ 215 kB

File size	Estimate*2	Measured (max)
4 MB	256	512
8 MB	128	128
16 MB	64	64
64 MB	16	16
128 MB	8	8
256 MB	4	4

300.000 documents @ 0.8 kB

File size	Estimate*2	Measured (max)
128 kB	512	1024
512 kB	128	256
1024 kB	64	64
2048 kB	32	32

300.000 documents @ 1.2 kB

File size	Estimate*2	Measured (max)
128 kB	1024	2048
512 kB	256	256
1024 kB	128	128
2048 kB	64	64

Table 5.1: Estimate of number of slotfiles compared to measured values using 10 buckets

5.5.1 Evaluation of estimate

Figure 5.10 and the corresponding Table 5.1 shows the estimated amount of slotfiles compared to the maximum amount of slotfiles per bucket in each test performed in Section 5.3. We observe a slight underestimation with the smallest slotfile sizes, while the estimate seems fairly correct on larger slotfiles. Clearly, adding a one slotfile split margin of error was a correct decision.

The underestimate with only smaller slotfile sizes makes sense. Large slotfiles mean a greater amount of documents in each file, increasing the probability of document size anomalies cancelling each other. This effect is gradually reduced while the slotfile size is increased, eventually causing at least one slotfile

to split an extra time.

Section 5.3 claims that the amount of slotfiles should not exceed 64. As we can see, none of the tests report an underestimate of 64 slotfiles or less. Thus, an underestimate is most likely to happen in cases where the amount of slotfiles is completely unsuitable. An underestimate at higher slotfile sizes than 64 is not a problem, since any value of 64 or above would nonetheless require the system to be reconfigured. Therefore, this estimate seems suitable for giving an impression of the suitability of the input settings.

Chapter 6

Tuning of MapReduce settings

In addition to VDS settings, certain configuration of the MapReduce implementation is possible. We will investigate these settings as well, in the same manner. The experiment methodology of Section 5.2 applies in this chapter as well.

6.1 Maximum size of MapReduce output documents

If any part of the output from a MapReduce operation is larger than what can be stored in a single document, it must be split into multiple documents. This creates additional overhead by increasing the amounts of Put operations, and also affects future reading of the data, adding the cost of another document retrieval.

Presumably, these documents should be as large as possible. However, both memory constraints and slotfile sizes could create an upper bound for the output document size. Since slotfile sizes are likely to be tuned for the documents normally stored in the system, the MapReduce output should adapt to this configuration.

The two-node cluster of Section 5.2.1 is utilised, using default VDS settings. It is then filled with 20.000 documents of average size 1.2 kB, to simulate a running system. Simulated MapReduce output documents are then placed in a single bucket.

The tests are performed by putting various amounts of random data into the cluster. To simulate token data, documents are only split into tokens of 76 byte each, causing documents to often be somewhat smaller than the stated maximum size, and possibly requiring an extra document for the remaining data to be stored. The test is performed using varying amounts of data, slotfile sizes and document sizes.

6.1.1 Results

Figure 6.1 shows the output of various tests runs with various settings. Unfortunately, the various node statistics do not show conclusive information, leaving this the only available information.

As can be seen in the figure, the maximum slotfile size appears to have little impact when putting 64 MB of data or less. When putting 128 MB or more into the same bucket, a peculiar peak appears in the middle of the graph. This is most noticeable with smaller slotfile sizes, but the larger slotfile sizes are also affected. One should not jump to conclusions with such limited material, but a combination of slotfile splits and the size of documents seems to be a plausible explanation, considering the information gathered in Section 5.3.1.

When using 32 or 64 MB document sizes, all requests seem to fail, since the amount of timeouts equal the total amount of documents. We also notice that 16 MB documents do *not* fail when using a maximum slotfile size of 8 MB. Instead, a single slotfile with the document in it is created, exceeding the size limit because it cannot be split. From this, it appears documents of size 32 MB or larger should not be stored in a VDS with a default request timeout, regardless of the slotfile size.

Upon further review of the test run with 256 MB of data, we notice that some requests always time out, regardless of the maximum slotfile size. We also see this tendency beginning at the stage of 128 MB data. From this and the fact that the initial amount of data in the system was relatively small, we believe that storing more than 64 MB of data into the same bucket should be avoided.

6.1.2 Suggested settings

Judging from the 32 and 64 MB graphs, Put performance is best with document sizes between 0.5 and 8 MB, regardless of the slotfile size. Considering that requests first fail at document sizes of around 1 MB when increasing the combined size of data, using a value between 1MB and 8MB, like for instance 4MB, appears to be a good default for overall use.

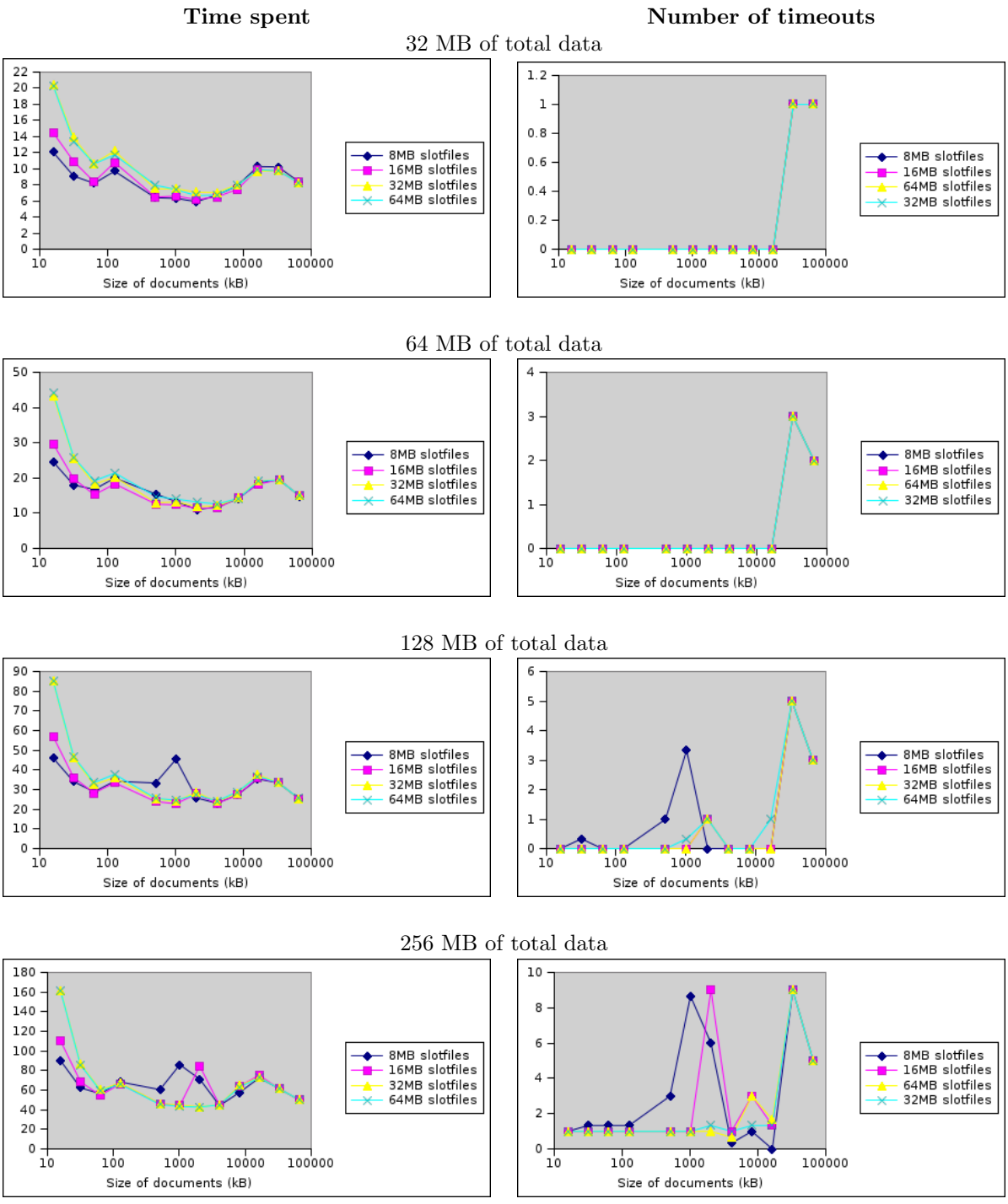

Figure 6.1: Time spent and number of timeouts when issuing Put requests with varying amounts of data, slotfile sizes and document sizes

6.2 Number of buckets for MapReduce output

As mentioned in Section 4.3, documents are stored into a configurable amount of buckets when output from the *map* and *reduce* operations. The amount of buckets have a number of effects.

First of all, the number of buckets represent an upper bound on the amount of nodes that will be used for further computation. This follows immediately from the fact that processing of the documents in a single bucket may not be split across nodes.

The number of buckets to use also affects the size of output buckets, as well as the amount of data in them. This means that a sufficiently large value should be used, in order not to overload any buckets with data.

6.2.1 Suggested settings

As discovered in Section 6.1, the amount of data stored per bucket in a VDS system with default settings should not exceed 64 MB. Also, a very large amount of documents per bucket is undesirable, meaning that splitting documents into multiple parts is undesirable.

The experiments also claim that 4 MB may be a feasible limit for the maximum document size. Considering that a maximum of 64 MB data is to be spread across at least as many documents per bucket as there are buckets in the system, the proposed limit of 4MB per document is very unlikely to be met. Realizing this, we focus on the total size of data stored in each bucket, and omit further discussion of maximum document size.

Provided that both existing data stored in buckets and MapReduce output is evenly spread across buckets, the minimum amount of buckets to be used for MapReduce output is:

$$\left\lceil \frac{MapReduce\ output\ size}{maximum\ bucket\ size - data\ already\ in\ buckets} \right\rceil$$

Since data is not necessarily evenly spread across buckets, this figure should be multiplied with some safety factor. What factor to use will depend on the particular system and MapReduce application. Keep in mind that increasing this factor too much will create a great amount of small documents spread across many buckets, which could reduce overall performance.

A problem arising from these features is MapReduce applications where the output data for a single key is larger than what will fit in a single bucket. There is no easy way around this. Compressing output data could improve the situation slightly, but only to a certain degree. Another approach may be to allow preprocessing of the tuples from an entire bucket before map output is saved[1], but this will also only work to a certain degree. Finally, the application could be implemented using multi-pass MapReduce [FMS+06], gradually merging the values. If the application is unsuitable for this approach as well, some program-specific solution must be found.

[1] This would also allow per-bucket aggregation in a straight-forward way.

6.3 Number of visitors per storage node

As described in section 3.6.4, visiting on a storage node uses a configurable amount of visitor instances. Since all processing in each thread is sequential, the best way to increase concurrent processing is to increase the number of concurrent visitors.

Unlike a normal visitor, the MapReduce visitor does not simply forward data to the client, but processes it on the node itself. The amount of processing is application-specific, and could in some cases require substantial amounts of time. Having other threads doing disk access while processing data might increase throughput.

Another difference from a normal visitor is how the MapReduce visitor returns data. Instead of returning data directly to the client, documents are saved by doing a substantial amount of asynchronous Put requests (see section 4.3). In order to guarantee that a complete data set has been stored, the visiting process must wait for each Put request to reply, causing the thread to sleep for a short period of time. Increasing the amount of threads should allow other processes to utilise resources while other visitors are sleeping.

Because of both the additional processing and the need to wait for Put replies, increasing the amount of visitors per storage node is likely to improve computing performance. However, too much concurrent processing has the cost of an increase in memory utilisation, thus limiting the memory available for keeping buffers and caching running. In extreme cases, only the visitors themselves would require more memory than available, causing computations to either fail or causing memory to be swapped to disk. This kind of memory starvation would very likely be disastrous for overall performance.

As indicated above, the number of threads should probably be kept as high as possible, but not exceed a certain limit. To avoid a too large memory footprint, measures should be taken to reduce the memory requirement of each thread. Such a possibility is reducing the number of buckets to visit simultaneously to a low value (like `1`), or reducing the size of document blocks.

Experiments were performed using the two-node cluster described in Section 5.2.1, with one storage node and all other services on the other node. The tests were performed with varying amounts of visitor threads, all with one bucket per visitor instance. All other settings were kept at default values. The system was populated with 1000 small documents containing location data from Yahoo! Local Search [Yah].

To perform the tests, three MapReduce applications were used. One application doing heavy CPU utilisation without storing data (Appendix B), one application for counting the number of occurrences of each city in the test data (Appendix C), and a combination of the two (Appendix D). Only the *map* step is performed in each test. The CPU-intensive applications were used both as-is and in a modified version where the CPU utilisation was tripled.

One may argue that a single storage node saving data back to itself is a suboptimal configuration, but bear in mind that an n times as large cluster would also imply n times as many MapReduce clients issuing Put requests. For the sake of simplicity and ease of monitoring, we therefore stick with a single node for this experiment.

6.3.1 Results

Figure 6.2 shows the time spent running the three MapReduce applications on a single-node cluster. As this figure clearly shows, adding more than one visitor is advantageous on both CPU and I/O-bound applications, but the gain of adding additional threads gradually decays. We discuss the results in more detail.

The purely I/O-dependent application shows improved performance when a second visitor thread is

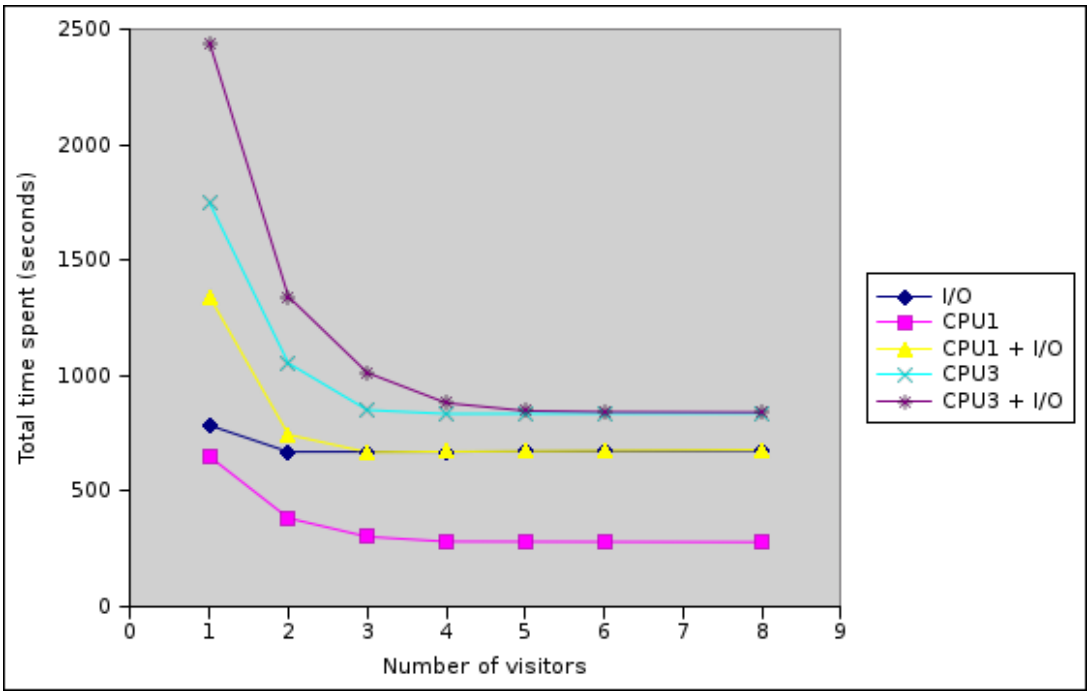

Figure 6.2: Time spent when running the *map* step of various MapReduce applications with varying amounts of visitor threads

added, but further threads do not make any difference. This is likely to be because this visitor does virtually no computing by itself, and that the application lifespan is primarily spent waiting for I/O. This has the effect of constantly having documents in queue for putting into VDS, keeping the operation bound to the performance of Put.

The purely CPU-bound implementations show gradual improvement until reaching 4 threads, but then stall. This behaviour is easily explained: On a dual CPU node, two constantly processing threads may very well utilise all available CPU. While two threads are processing, the others buffer data. Since the processing in this case is very time-consuming, a single extra thread for each CPU is sufficient for buffering. Two threads per CPU is necessary because threads may not migrate between CPUs [Sta01], thus requiring two dedicated threads for each CPU.

We notice that with combined CPU and I/O load where the I/O is the most time-consuming, the combined and I/O-specific plots show an equal time spent when using four threads. Since writing the data is the most time-consuming, a single thread has time to receive and process data while another thread is writing. To allow full utilisation of both CPUs for processing, one set of threads must exist for each CPU, giving a total of four threads.

With combined CPU and I/O load where the CPU is more time-consuming, the gradual improvement in runtime continues until adding 5 threads, although the difference between 4 and 5 is quite small. At 5 threads, the combined and purely CPU-bound tests complete equally fast, showing that computations are never forced to wait for I/O requests to complete. At first glance, one would expect these results to show optimal performance already at 4 threads, like the other CPU-bound results. However, keep in mind that a thread does not normally have exclusive CPU privileges while running [Sta03]. This allows two computations to complete at the same time, causing both threads to be waiting for I/O. According to these results, a single spare thread is sufficient for resolving such issues in a dual CPU setup.

6.3.2 Suggested settings

The experiment results show gradually improving performance when increasing the amount of threads in a dual CPU system until reaching 5 threads, but additional threads appear to have no effect. The discussion indicates that 2.5 threads per processor or less is also sufficient in a system with more than two CPUs.

Since the number of visitor threads is set to the same value for all nodes being visited, a common value suitable for all nodes must be selected, even if the nodes do not consist of the same number of CPUs. CPU-wise, a few extra threads do not seem to affect performance. Therefore, a value suitable for the node with the most CPUs should be selected. Where c_{max} is the amount of CPUs in the node with the most CPUs, the optimal number of threads t_{CPU} is:

$$t_{CPU} = \lceil 2.5 * c_{max} \rceil$$

Also, there is the possible issue of memory starvation. As Section 4.3.4 describes, for each bucket currently being processed using MapReduce, all tuples created from the bucket must be kept in memory. While this is not normally expected to be very much data, situations might arise where memory could become an issue. With one bucket per visitor the maximum amount of visitors is where the combined memory for each thread will fit in the memory of the node with least available memory.

Building upon the equation presented in Section 4.3.4 and remembering that the number of visitor threads applies to all storage nodes, we may express an upper bound t_{memory} for the amount of visitor threads fitting in memory on a single node:

$$\begin{aligned} t_{memory} &= \left\lfloor \frac{m_{node}}{\frac{|D|}{|B|} * s} \right\rfloor \\ t_{memory} &= \left\lfloor \frac{m_{node} * |B|}{|D| * s} \right\rfloor \end{aligned}$$

Where m_{node} is the available memory on the node, D is the set of documents, B is the set of buckets, and s is the combined size of all tuples being returned from processing a single document.

When running MapReduce, a value equal to or below both t_{CPU} and t_{memory} should be chosen. However, other factors also apply, possibly requiring a lower setting than what presented above. First of all, there is the issue of concurrent processing. In a deployed system, using absolutely all available resources on storage nodes may reduce the performance of other applications. In such cases, the number of threads may be reduced to give other applications priority.

6.4 Throttling of MapReduce bandwidth utilisation

Since a MapReduce run will issue a great amount of Put requests while storing data back into VDS, there is a risk of network overloading. Since MapReduce will typically be run to aggregate data in a production environment, the MapReduce computation should be careful not to obstruct normal operations via excessive bandwidth utilisation. Depending on the urgency of the computations, one could specify to utilise only parts of the available bandwidth, all available bandwidth, or even more than the available bandwidth (at the cost of slowing down other operations).

While there is currently no specific configuration option for throttling of MapReduce, this would be possible by only visiting a subset of the buckets at a time, and adjusting how fast visiting of other buckets is requested. Since each *map* or *reduce* instance will read from separate buckets, and also store into multiple buckets, there is a fair chance of even network utilisation when visiting a limited amount of buckets at a time. Thus, limiting the amount of buckets to visit in a given time frame should allow throttling of the system-wide bandwidth utilisation of MapReduce.

Because the MapReduce implementation returns data using Put requests, Put requests are expected to amount to most of the bandwidth requirements for MapReduce. The visiting API will only issue up to one *createVisitor* request per bucket and receive the associated replies, which is not expected to be even remotely close to the amount of data stored from a bucket. We therefore ignore visiting-related bandwidth requirements, and focus solely on the bandwidth requirements of Put.

6.4.1 Expected results

The total amount of network traffic is given by the sum of all messages passed during a request. These messages include passing the message to the distributor, passing the message to the first storage node, and all replies. If more than one copy is to be stored, one additional message and reply per additional copy is added. We omit the *getSystemState* request and reply, as these are only occasionally performed.

Given a message header size m, average document payload size d and number of copies n, the average total network traffic T_{Put} is given by:

$$\begin{aligned} T_{Put} &= \textit{Message to distributor and reply} + \\ &\quad \textit{Messages to storage nodes and replies} + \\ T_{Put} &= ((m+d)+m) + n*((m+d)+m) \\ T_{Put} &= (n+1)*(2m+d) \end{aligned}$$

Where m is about 200 bytes, d depends on the documents being stored and n depends on the *numberofcopies* configuration item.

Note however, that the above calculations do not include network overhead. An Ethernet packet is typically 64 to 1518 bytes long, depending on the payload size. This includes a 14 byte MAC header and a 4 byte CRC checksum [IEE05]. In addition, 40 bytes of IP and TCP headers are required for TCP traffic [DAR81a, DAR81b], leaving 1460 bytes for data and 58 bytes for headers and the checksum. This gives an overhead of

$$\left\lceil \frac{\textit{data size}}{1460} \right\rceil * 58$$

per message transmitted, giving the following estimate of total bandwidth utilisation:

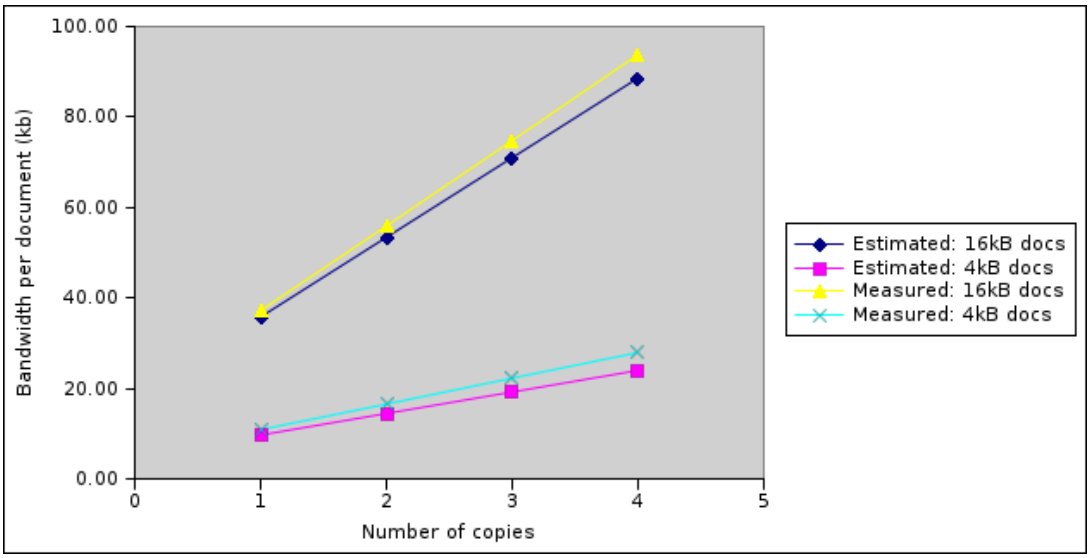

Figure 6.3: Average bandwidth required for putting documents into VDS, estimated and measured values

$$T_{Put} = (n+1)*(2m+d)+(n+1)*\left(\left\lceil\frac{m}{1460}\right\rceil+\left\lceil\frac{m+d}{1460}\right\rceil\right)*58$$

6.4.2 Experiment

To verify the correctness of the above bandwidth analysis, we will perform an experiment and compare the results. To perform this experiment, nine nodes from the 10-node cluster of Section 5.2.1 was reconfigured with the following setup:

- One client
- One fleet controller
- One distributor
- Six storage nodes

By separating all services on separate nodes, we will be able to monitor the total bandwidth utilisation by combining the measured network traffic on each node.

Experiments were performed with 25.000 documents, in two tests, one with 4kB documents, another with 16kB documents. The documents were filled with an exact amount of random characters each, and issued document IDs with an average length of 20 bytes.

Results

Figure 6.3 shows a plot of the expected bandwidth utilisation using various settings for number of copies, alongside the average per-document bandwidth utilisation of an entire VDS system when putting 25.000 documents. The same results are included in table 6.1, along with some additional information. Note that the measured values also include all other network traffic at the time, including the fleet controller,

Document size	Time spent	Expected bandwidth	Measured bandwidth	Difference in kb	Difference in percent
4 kB	0.101 s	9.73 kb	10.92 kb	1.19 kb	10.93%
4 kB	0.102 s	14.48 kb	16.56 kb	2.09 kb	12.60%
4 kB	0.104 s	19.22 kb	22.27 kb	3.05 kb	13.69%
4 kB	0.106 s	23.97 kb	27.93 kb	3.96 kb	14.17%
16 kB	0.100 s	35.87 kb	37.27 kb	1.40 kb	3.76%
16 kB	0.100 s	53.43 kb	55.95 kb	2.53 kb	4.51%
16 kB	0.100 s	70.99 kb	74.77 kb	3.79 kb	5.06%
16 kB	0.102 s	88.54 kb	93.73 kb	5.18 kb	5.53%

Table 6.1: Average data collected when putting 25.000 documents into VDS.

logging and other non-VDS network activity. Thus, measured values are expected to be somewhat higher than estimated values.

From Table 6.1, we observe that the difference in size between estimated and measured values appears to be near constant, only slightly increasing with document size. They do show a greater difference as the number of copies is increased, which might be related to the increase in the amount of transmissions. This may be caused by multiple reasons, like for example network packet retransmissions. Instead of investigating this matter further, we will simply recommend adding some margin of error to the bandwidth estimate.

6.4.3 Suggested settings

As just discovered, the total bandwidth utilisation of Put may be computed using an estimate of the average size of documents, along with an estimate of the amount of documents. After computing these values, one may use them for throttling of bandwidth in MapReduce operations. By specifying the desired amount of bandwidth MapReduce should be allowed to use, one could add a suitable delay between the visiting of each batch of buckets, guaranteeing that the entire MapReduce computation will require at least a certain amount of time.

The delay between each bucket could be computed using the following function, where t_{delay} is the delay in seconds, T_{tot} is the expected total network traffic for all data, $R_{MapReduce}$ is the bit rate MapReduce may use, and B is the set of buckets for the computation:

$$t_{delay} = \frac{T_{tot}}{R_{MapReduce} * |B|}$$

This interval could either be computed prior to running the application, and possibly updated during the MapReduce run, should the value of $R_{MapReduce}$ change. This could be the case in a setting where the system load is expected to vary over time, allowing more bandwidth to be used by MapReduce when other clients are idle.

Chapter 7

Future work

During the course of the thesis work, a number of possibilities for future work have been discovered. This includes both research outside the scope of this thesis and extensions of the work done. We provide a brief list of suggestions.

In a dynamically growing storage system, the volume of data may grow larger than originally anticipated. The amount of buckets in VDS is only configurable before deployment of the VDS installation. This requires the administrator to anticipate the total data volume to be stored in the VDS instance for all future, risking that a wrong estimate will cause problems when the system grows larger. This may be an ability to simply change the number of buckets, but a tree-based approach similar to the one used in the CRUSH algorithm presented in Section 2.2.5 may also be considered.

Although bandwidth considerations when running MapReduce concurrently with other VDS applications has been discussed, time and resources did not allow a complete analysis of the combined effect of multiple different applications concurrently using a running VDS system. Further work should include an analysis of the performance of VDS during a variety of concurrent operations, both reading and storing data at the same time. The call frequency of each operation should also be considered, possibly allowing tuning based on usage patterns. An analysis of various VDS operations is included in Appendix E, and may be used as a starting point for further research.

Section 6.2 mentions a possible enhancement to the MapReduce library, with pre-processing of map output tuples after the *map* step, allowing per-bucket aggregation before data is saved. Depending on the application, this may cause a substantial reduction in the data volume, both reducing data transfer between nodes, the size of target buckets, and the amount of computation in the *reduce* step. This is a good candidate for further research and possibly implementation.

The size of output data it not only a problem during data transfer and storage. For applications with substantial memory requirements, available memory on the node may become an issue, as briefly mentioned in Section 6.3.2. A well-known technique in distributed database systems is to handle memory exhaustion by transferring data prematurely to the destination node [Bra03], at a possible cost of increased bandwidth requirements (since local and previously transferred data may not be merged). This may also be combined with the preprocessing mentioned above, depending on the associativity and commutativity of the performed operations. Research of this possibility and possible automated enabling of it could prove worthwhile.

The possibility of throttling the bandwidth requirements of MapReduce by delaying visiting of buckets is described in Section 6.4.3. However, this seemingly simple enhancement depends on knowledge of available bandwidth resources throughout the system. Thus, an efficient method for measuring the currently available bandwidth is necessary. In fact, this method could be extended to retrieving other load-related state from the nodes as well, allowing visiting to be scheduled to the least busy node. Such state may

also be used for adjusting the number of threads on a per-bucket basis, allowing maximum utilisation of spare capacity, while still behaving on more busy nodes. Further research on these subjects could allow heavy MapReduce computations to be performed purely by utilisation of otherwise underutilised nodes, placing a minimal burden on the system.

Chapter 8

Experiences

While working with this thesis, a few slight challenges and inconveniences were encountered. We give a brief overview, along with how the issues were handled.

Refactoring of the relevant *map* and *reduce* prototypes from [Gry06] into a complete MapReduce library was expected to be a straight-forward process. Unfortunately, these prototypes were developed while VDS was still in an early stage of development. In the meantime, a lot of VDS APIs and associated libraries had undergone numerous changes, turning the refactoring process to a more time-consuming process than expected.

An important and major part of the thesis work was the creation of VDS documentation. Being a fairly new and developing product, VDS documentation is still somewhat sparse. To be able to do proper tuning of VDS, a good understanding of the inner workings was necessary. A substantial amount of time was therefore spent inquiring VDS developers on implementation details of the various VDS features and producing the necessary documentation. The documentation was later proofread by one of the developers, to ensure correctness.

During development of the hardware monitoring framework, the use of existing Munin [Mun] plugins saved a lot of work. Some studying of the Munin source code was required to create the framework itself, but with previous RRDTool [Oet] experience, creating this simple framework was quite straight-forward. However, Munin did not provide an `iostat` plugin for FreeBSD, only for Linux. Naturally, this plugin was missing for a good reason. The FreeBSD 4.x iostat implementation proved to be very unsuitable for use by Munin plugins, in spite of having seemingly suitable command-line options in the documentation. These options were simply ignored in the FreeBSD 4.x version of iostat, but do for the record behave as expected in FreeBSD 6. Eventually, the iostat monitoring was implemented by creating a wrapper daemon around iostat, creating a text file with the same format as the originally planned Munin plugin.

Some issues were also encountered regarding allocation of experiment hardware. A small cluster of nodes located in the US were swiftly allocated, and all seemed well. However, the nodes kept disabling the login account a few minutes after creation, making access to the cluster impossible. After substantial troubleshooting of this issue, two local nodes were allocated, and experiments only requiring two nodes could commence. The remainder of the experiments were forced to hold until quite close to deadline, when the login issues were finally resolved. In retrospect, hardware allocation should have been attempted earlier, and not postponed until the hardware was required for further progress. Because of the lack of available hardware, a few experiments had to be sacrificed in order to reach the thesis deadline. This includes the verification of some of the VDS analysis provided in Appendix E. These experiments have instead been proposed as future work.

Since putting documents into VDS can be quite time-consuming, a VDS developer had created an application for creating slotfiles directly on the storage nodes, allowing a VDS system to quickly be

populated with random data. The application was tested on the two-node cluster and found to be working well. However, when switching to a larger cluster and requiring significantly more data, the application started failing. The application was simply designed for a slightly different use-case, and patching it proved to be a complicated matter. Since the developer did not have time for this at the moment, all visiting experiments instead had to rely on data being entered using conventional Put requests. This is the main reason why the experiments in Section 5.4 were only performed with empty buckets.

Although a few technical issues arose, it is worth mentioning that the impact of these problems were greatly reduced by the help of Yahoo! employees, who did their very best to help resolve all issues encountered, as well as providing helpful advice for troubleshooting and workarounds.

Chapter 9

Conclusion

We have described the functionality of VDS, MapReduce and the design of the VDS MapReduce framework. We have also described other file systems which are related to VDS or MapReduce in various ways.

We have developed a working MapReduce framework based on prototypes from previous work, and described the design of it. A framework for monitoring various metrics on VDS nodes during performance testing has also been developed. Both frameworks have been used in multiple experiments.

Several configuration items, both general VDS settings and MapReduce specific options, have been researched. The research uses a variety of methods, depending on the characteristics of each configuration item. We have examined results, and proposed either generic defaults, algorithms for automated configuration, or guidelines for administrators.

During the research, we have also discovered a variety of subjects for future study. We have provided a list of the most promising.

Bibliography

[BBDW83] Dina Bitton, Haran Boral, David J. DeWitt, and W. Kevin Wilkinson. Parallel algorithms for the execution of relational database operations. *TODS*, 1983.

[Bra03] Kjell Bratsbergsengen. *TDT4225 Lagring og behandling av store datamengder.* Tapir Akademisk Forlag, 2003.

[DAR81a] DARPA. *RFC 791: Internet Protocol, DARPA Internet Program, Protocol Specification*, 1981.

[DAR81b] DARPA. *RFC 793: Transmission Control Protocol, DARPA Internet Program, Protocol Specification*, 1981.

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

[Fel06] Jon Feldman. Using many machines to handle an enormous error-correcting code. *Proc. IEEE Information Theory Workshop*, 2006.

[FMS+06] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Cliff Stein, and Zoya Svitkina. *On the Complexity of Processing Massive, Unordered, Distributed Data*, 2006. `http://arxiv.org/abs/cs.CC/0611108`.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SOSP*, 2003.

[GMUW02] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems – The Complete Book.* Prentice Hall, 2002.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, 1993.

[Gry06] Knut Auvor Grythe. Implementing mapreduce using vespa document store, 2006.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[HM] R. J. Honicky and Ethan L. Miller. *RUSH: Balanced, Decentralized Distribution for Replicated Data in Scalable Storage Clusters.* `http://www.cse.ucsc.edu/~honicky/rush_tech_report.pdf`.

[HM03] R. Honicky and E. Miller. A fast algorithm for online placement and reorganization of replicated data, 2003.

[HM04] R. Honicky and E. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution, 2004.

[IEE05] IEEE. *IEEE Std 802.3^{TM}-2005: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*, 2005.

[Läm06] Ralf Lämmel. *Google's MapReduce Programming Model – Revisited*. Microsoft Corp., 22 January 2006. `http://www.cs.vu.nl/~ralf/MapReduce/`.

[Lin] Linpro. *The Linpro Website*. `http://linpro.no/`.

[Mun] The Munin Project. *The Munin Website*. `http://munin.projects.linpro.no/`.

[Oet] Tobi Oetiker. *RRDTool*. `http://oss.oetiker.ch/rrdtool/`.

[Pet90] C. Peterson. Parallel distributed approaches to combinatorial optimization: Benchmark studies on traveling salesman problem. *Neural Computation*, 2(3):261–269, 1990.

[PM90] J. F. Pekny and D. L. Miller. A parallel branch and bound algorithm for solving large asymmetric traveling salesman problems. *Proceedings of the 1990 ACM annual conference on Cooperation*, 1990.

[RLA00] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *USENIX Conf. Proc., June 2000*, pages 41–54, 2000.

[Ses92] S. Seshadri. *Probabilistic methods in query processing*. PhD thesis, Madison, WI, USA, 1992.

[SM82] Stanley Y. W. Su and Krishna P. Mikkilineni. Parallel algorithms and their implementation in micronet. In *VLDB '82: Proceedings of the 8th International Conference on Very Large Data Bases*, pages 310–324, San Francisco, CA, USA, 1982. Morgan Kaufmann Publishers Inc.

[SN95] Ambuj Shatdal and Jeffrey F. Naughton. Adaptive parallel aggregation algorithms. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 104–114, New York, NY, USA, 1995. ACM Press.

[Sta01] William Stallings. *Operating Systems*. Prentice Hall International, Inc., fourth edition, 2001.

[Sta03] William Stallings. *Computer Organization & Architecture*. Prentice Hall International, Inc., sixth edition, 2003.

[Sze] M. Szeredi. *File System in User Space*. `http://fuse.sourceforge.net/`.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrel D. E. Long, and Carlos Maltzahn. *Ceph: A Scalable, High-Performance Distributed File System*, 2006.

[WBMM06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006. `http://www.ssrc.ucsc.edu/Papers/weil-sc06.pdf`.

[Yah] Yahoo! Inc. *Yahoo! Local Search*. `http://yahoo.com/`.

Appendix A

The no-op MapReduce application

```
#include "storage/visiting/visitors/mapreducevisitor.h"

namespace storage {
class MapReduceNoOp : public MapReduceVisitor {

public:
    MapReduceNoOp(StorageServerInterface& server,
                  const api::Parameters& params)
                  : MapReduceVisitor(server, params)
    {
    }

private:
    void map(std::auto_ptr<document::Document> doc);
    void reduce(std::string key, std::vector<std::string> values);
};
}

LOG_SETUP(".mapreducenoop");
REGISTER_VISITOR(storage::MapReduceNoOp);

using namespace storage;

void MapReduceNoOp::map(std::auto_ptr<document::Document> doc)
{
    // Discard input and do nothing
    (void) doc;
}

void MapReduceNoOp::reduce(std::string key, std::vector<std::string> values)
{
    // Discard input and do nothing
    (void) key;
    (void) values;
}
```

Appendix B

The CPU-intensive MapReduce application

```
#include "storage/visiting/visitors/mapreducevisitor.h"
#include <math.h>

namespace storage {
class MapReduceHeavyCPU : public MapReduceVisitor {

public:
    MapReduceHeavyCPU(StorageServerInterface& server,
                      const api::Parameters& params)
                      : MapReduceVisitor(server, params)
    {
    }

private:
    void map(std::auto_ptr<document::Document> doc);
    void reduce(std::string key, std::vector<std::string> values);
};
}

LOG_SETUP(".mapreduceheavycpu");
REGISTER_VISITOR(storage::MapReduceHeavyCPU);

using namespace storage;

void MapReduceHeavyCPU::map(std::auto_ptr<document::Document> doc)
{
    // Discard input
    (void) doc;

    // Do CPU-intensive calculations
    for (int i=0; i<40000000; i++) {
        sqrt(i);
    }
}

void MapReduceHeavyCPU::reduce(std::string key, std::vector<std::string> values)
{
```

```
    // Discard input and do nothing
    (void) key;
    (void) values;
}
```

Appendix C

The MapReduce application for counting cities

```
#include "storage/visiting/visitors/mapreducevisitor.h"
#include <time.h>

namespace storage {
class MapReduceCityCount : public MapReduceVisitor {

public:
    MapReduceCityCount(StorageServerInterface& server,
                       const api::Parameters& params)
                       : MapReduceVisitor(server, params)
    {
    }

private:
    void map(std::auto_ptr<document::Document> doc);
    void reduce(std::string key, std::vector<std::string> values);
};
}

LOG_SETUP(".mapreducecitycount");
REGISTER_VISITOR(storage::MapReduceCityCount);

using namespace storage;

void MapReduceCityCount::map(std::auto_ptr<document::Document> doc)
{
    // Emit name of city
    try {
        const document::Field& currField = doc->getField("city");
        std::auto_ptr<document::FieldValue> currFieldValue(doc->getValue(
            currField));
        emit(currFieldValue->serializeText(), "1");
    } catch (document::FieldNotFoundException e) {
        (void) e; // not a city, ignore
    }
}
```

```
void MapReduceCityCount::reduce(std::string key, std::vector<std::string> values)
{
    // Combine into a single sum
    int n = 0;
    for (std::vector<std::string>::iterator i = values.begin(); i!=values.end();
        i++) {
        n += atoi(i->c_str());
    }
    emit(key, vespalib::make_string("%d", n));
}
```

Appendix D

The CPU-intensive MapReduce application for counting cities

```
#include "storage/visiting/visitors/mapreducevisitor.h"
#include <math.h>

namespace storage {
class MapReduceHeavyCity : public MapReduceVisitor {

public:
    MapReduceHeavyCity(StorageServerInterface& server,
                       const api::Parameters& params)
                       : MapReduceVisitor(server, params)
    {
    }

private:
    void map(std::auto_ptr<document::Document> doc);
    void reduce(std::string key, std::vector<std::string> values);
};
}

LOG_SETUP(".mapreduceheavycity");
REGISTER_VISITOR(storage::MapReduceHeavyCity);

using namespace storage;

void MapReduceHeavyCity::map(std::auto_ptr<document::Document> doc)
{
    // Do CPU-intensive calculations
    for (int i=0; i<40000000; i++) {
        sqrt(i);
    }

    // Emit name of city
    try {
        const document::Field& currField = doc->getField("city");
        std::auto_ptr<document::FieldValue> currFieldValue(doc->getValue(
            currField));
        emit(currFieldValue->serializeText(), "1");
```

```
    } catch (document::FieldNotFoundException e) {
        (void) e; // not a city, ignore
    }
}

void MapReduceHeavyCity::reduce(std::string key, std::vector<std::string> values)
{
    // Combine into a single sum
    int n = 0;
    for (std::vector<std::string>::iterator i = values.begin(); i!=values.end();
        i++) {
        n += atoi(i->c_str());
    }
    emit(key, vespalib::make_string("%d", n));
}
```

Appendix E

Cost analysis of various VDS operations

During the thesis work, the costs of various VDS operations were examined. However, due to limited time and resources, all were not used or verified. They are left here as material for further work.

E.1 Put

We discuss the cost of a Put request in terms of bandwidth, disk access and memory utilisation.

E.1.1 Disk access

The Put procedure consists of reading the slotfile header and metadata list, which is all stored sequentially on disk. Then the document header, body and metadata list entry are written to three different positions on disk. This gives a minimum of one read and three writes on each node storing the bucket.

However, this is not always the case. When doing the initial read of the document header, additional data may also have been read. This could include the entire metadata list, the document headers and even the position of the initial document. In such cases, the data is updated in memory and all data is written in one large write, to reduce the amount of seeking. This might result in as little as one read and one write, although two writes is a more likely result. The size of the initial slotfile read is configurable.

Note that the disk cache of the underlying file system also comes into play, further reducing disk activity. This may be overridden by configuring that an explicit file sync should be performed after each operation.

E.1.2 Memory utilisation

The client and distributor only uses memory for buffering requests and replies, so their memory requirements can easily be extracted from section 6.4.1. We will therefore focus on the minimum memory requirements on the storage node.

On the storage node, the incoming document and the slotfile parsing requires memory. Given an average document size d, slotfile header size h, slotfile metadata list entry size e and slotfile metadata list length n, the minimum memory utilisation M_{Put} for a document with average size is given by:

$$\begin{aligned} M_{Put} &= \textit{Incoming document} + \textit{Slotfile metadata} \\ M_{Put} &= d + (h + n * e) \\ M_{Put} &= d + h + n * e \end{aligned}$$

Where h is 20 bytes, e is 40 bytes, and d and n are application dependent.

Note that this is a minimum. In actuality, a lot of memory is spent in preallocated buffers. If the Put requires the slotfile to be enlarged or split, this will also require memory.

E.2 Remove

Due to the fact that a Remove is quite similar to a Put, the cost of a request is computed in a similar manner.

E.2.1 Bandwidth

The bandwidth utilisation of a Remove request equals the cost of a Put (see 6.4.1), except that no document content is transmitted.
Given a message header size m and number of copies c, the total network traffic T_{Remove} is given by:

$$\begin{aligned} T_{Remove} &= \textit{Message to distributor and reply} + \\ &\quad \textit{Messages to storagenodes and replies} \\ T_{Remove} &= (m + m) + n * (m + m) \\ T_{Remove} &= 2m + c * (2m) \\ T_{Remove} &= (2c + 2) * m \end{aligned}$$

Where m is about 200 bytes and c depends on the *numberofcopies* configuration item.

E.2.2 Disk access

Removing documents from VDS is done by writing a new entry in the metadata list, stating that the document is deleted. This procedure consists of reading the slotfile header, metadata list and in most cases a lot of the document headers, all stored sequentially on disk. As soon as a list entry with a hash matching the correct document ID is located, the document metadata block for this entry is read and the document ID is verified. This check is necessary to avoid any (very unlikely) occurrences of multiple document IDs in the same slotfile sharing the same hash. After verifying the ID the new metadata list entry is written to disk.

This gives a minimum of one read and one write on each node containing the bucket. In cases where the document headers were not read along with the metadata list, another read is necessary. Operating system I/O caching might further reduce the amount of file I/O.

E.2.3 Memory utilisation

The client and distributor only uses memory for buffering requests and replies, so their memory requirements can easily be extracted from section E.2.1. We will therefore focus on the minimum memory requirements on the storage node.

On the storage node, the incoming request, the slotfile parsing requires memory. Given a slotfile header size h, slotfile metadata list entry size e and slotfile metadata list length n, the minimum memory utilisation M_{Remove} is given by:

$$\begin{aligned} M_{Remove} &= \textit{Slotfile header} + \textit{Slotfile metadata list} \\ M_{Remove} &= h + m * e \end{aligned}$$

Where h is 20 bytes, e is 40 bytes, and n is application dependent.

Also note that a lot of memory is used in pre-allocated buffers. The figures presented here is a minimum.

E.3 Get

The cost of a Get request is somewhat similar to a Put request, except that only one storage node is involved.

E.3.1 Bandwidth

The bandwidth cost of a Get request equals a request to the distributor, a request to one of the storage nodes containing the document, and replies for these requests. We omit the getSystemState call, since this is rarely done.

Given a message header size m, getSystemState reply payload size s and average document payload size d, the average total network traffic T_{Get} is given by:

$$\begin{aligned} T_{Get} &= \textit{Message to distributor and reply} + \\ &\quad \textit{Messages to storagenodes and replies} \\ T_{Get} &= (m + (m + s)) + (m + (m + d)) + (m + (m + d)) \\ T_{Get} &= (2m + d) + (2m + d) \\ T_{Get} &= 4m + 2d \end{aligned}$$

Where m is about 200 bytes and d depends on the documents being stored.

E.3.2 Disk access

The get procedure consists of reading a chunk of the slotfile consisting of the slotfile header, metadata list and possibly the document headers, all stored sequentially on disk. Then the document header (if not part of the initial read) and body is read from disk.

This will in most cases cause two or three reads on one of the nodes containing the bucket. Also note that the metadata is likely to be cached, but not the document body. If this is the case, only one read

will be performed.

E.3.3 Memory utilisation

The client and distributor only uses memory for buffering requests and replies, so their memory requirements can easily be extracted from section E.3.1. We will therefore focus on the memory requirements on the storage node.

On the storage node, the incoming request, the reply and the slotfile parsing requires memory. Given an average document size d, slotfile header size h, slotfile metadata list entry size e and slotfile metadata list length n, the average memory utilisation M_{Get} is given by:

$$\begin{aligned} M_{Get} &= \textit{Requested document} + \textit{Slotfile metadata} \\ M_{Get} &= d + (h + n * e) \\ M_{Get} &= d + h + n * e \end{aligned}$$

Where h is 20 bytes, e is 40 bytes, and d and n are application dependent.

E.4 Visit

Determining the cost of a Visit request is slightly more complicated, as the requests in a greater degree depend on configuration options. The cost of a request will as we will see depend on the amount of document blocks returned to the client.

E.4.1 Bandwidth

The bandwidth requirement of a visiting request consists of the following:

- one *getBucketNodes* request for each relevant distributor
- one *createVisitor* request for each relevant node
- a number of document blocks returned to the client
- replies for all requests

If all buckets are to be visited and $|B| * c \gg |S|$, where B is the set of buckets, c is the configured number of copies and S is the set of storage nodes, we assume that buckets are evenly distributed across storage nodes. In this case, a visitor is likely to be created on every node, and the amount of document blocks will depend on the amount of documents, the document size and the document block size. Given the set D of documents of average size d and a document block size b, the number of document blocks n is given by:

$$n = \left\lceil \frac{|D| * d}{b} \right\rceil$$

If visiting small amounts of buckets, the probability of an even distribution decreases. Nonetheless we will keep assuming they are evenly distributed for the sake of simplicity.

Given a message header size m, *getBucketNodes* reply size β, *createVisitor* payload size v, set of relevant distributors Ψ, set of relevant storage nodes S, average document payload size d and docblock size b, the average total network traffic T_{Visit} is given by:

$$
\begin{aligned}
T_{Visit} &= \textit{getBucketNodes requests and reply} + \\
&\quad \textit{createvisitor calls for each node and reply} + \\
&\quad \textit{Document blocks} \\
T_{Visit} &= |\Psi| * (m + (m + \beta)) + |S| * ((m + v) + m) + \left\lceil \frac{|D| * d}{b} \right\rceil * b \\
T_{Visit} &= 2|\Psi|m + 2|S|m + |\Psi|\beta + |S|v + \left\lceil \frac{|D| * d}{b} \right\rceil * b \\
T_{Visit} &= (|\Psi| + |S|) * 2m + |\Psi|\beta + |S|v + \left\lceil \frac{|D| * d}{b} \right\rceil * b
\end{aligned}
$$

Where m is about 200 bytes and all other values are application-dependent.

E.4.2 Disk access

Since a visit will read all slotfiles in the buckets being visited, the amount of disk access depends on the amount of slotfiles. The amount of slotfiles depends on the configuration item *maxfilesize* and a few other limits which we will ignore for the sake of simplicity.

Given a bucket λ, the amount of documents in it n_λ, the average document size d, the *maxfilesize* setting and some positive integer x, the amount of slotfiles N_λ is at least

$$N_\lambda = 2^x \geq \left\lceil \frac{n_\lambda * d}{maxfilesize} \right\rceil$$

Note that this is a lower bound. The actual figure might be larger, depending on how well each slotfile can be filled. Documents are never split between slotfiles, and the total space required by a set of documents will rarely fit exactly to the maximum slotfile size.

When a server-side visitor initiates visiting of a bucket, it first opens all slotfiles and determines which documents should be visited. This is done by reading the header and metadata list of each slotfile. If a document selection string is provided, the header blob for each document must also be read. The header blobs are read in the physical order they have on disk to allow for long, sequential reads. This gives a constant cost of at least one read per slotfile, or two if using a document selection string.

After determining which documents to visit, visiting commences. When filling one document block with data, the slotfile header, metadata list, document headers and document body is read. This requires at least one read for the slotfile header and metadata list, at least one read for the document headers, and at least one read for the document bodies. If all documents in the slotfile do not fit in the docblock, the remaining must wait for the next docblock.

This gives a minimum of three reads per docblock, with $\left\lceil \frac{maxfilesize}{docblock\ size} \right\rceil$ document blocks per slotfile.

E.4.3 Memory utilisation

The visiting process utilises a number of lists and buffers on the storage node, most of which are affected by configuration options. These include:

- A list of remaining buckets to visit
- One document block per persistence manager thread
- One list of pending documents per persistence manager thread

The number of items in the list of remaining buckets on each storage node is initially equal to the amount of buckets being visited on it. Assuming the set B of buckets requested for visiting by the client is evenly distributed across the set S of all storage nodes. In that case, $\left\lceil \frac{|B|}{|S|} \right\rceil$ is the initial number of elements in the list of remaining buckets.

The number of document blocks stored in memory is equal to the number of threads in the system, which is equal to the number of disks multiplied by the number of threads per disk.

The number of items in the lists of pending documents depend on the amount of documents being visited at a time. Assuming an even distribution of documents, this is a function of the amount of buckets being visited simultaneously. Given the set D of all documents being visited on all nodes, the set B of all buckets being visited and the maximum amount a of buckets to visit simultaneously on a node, the combined length of all pending documents lists on the node will be $\left\lceil \frac{|D|}{|B|} \right\rceil * a$.

Summarising, given the set D of all documents being visited on all nodes, the set B of all buckets being visited, the size r of an entry in the remaining bucket list, the number h of hard disk drives on each storage node, the number t of threads per disk, the size d of a document block, the maximum amount a of buckets to visit at once and the size p of an entry in the pending documents list, the total memory consumption M_{Visit} on a storage node during visiting is

$$M_{Visit} = \left\lceil \frac{|B|}{|S|} \right\rceil * r + h * t * d + \left\lceil \frac{|D|}{|B|} \right\rceil * a * p$$

To ensure that the network throughput is always maximised, storage nodes buffer as many messages as possible in main memory. On nodes where the disk I/O outperforms network throughput (a quite likely scenario on a node with multiple disks with decent performance), these buffers are likely to fill. The size of these buffers is limited by the configuration setting `maxbuffersize`.

E.5 Operation-independent costs

Certain parts of VDS require a constant amount of resources, independently of issued requests. Although constant, these requirements are configuration-dependent, and tuning of this configuration could possibly allow other parts of VDS to allocate additional resources. Examples of such requirements include:

- The bucket database on the distributor nodes
- Updates of the system state