Partitioning an Open Source Database in the Cloud

Even Østvold

Master of Science in Informatics
Submission date: June 2014
Supervisor: Svein Erik Bratsberg, IDI

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

Summary

Cloud computing has become a part of how web applications are being developed. The easy access to virtually endless supply of servers that the cloud provides has brought many benefits, but at the precondition that the application is able to utilize its distributed nature.

Data storage is an integral part of most applications and most cloud providers offers services to facilitate storage. Some users of the cloud may however decide to use a database solution of their own choice.

In this thesis we will look at how a NoSQL database, MongoDB, may be implemented in Windows Azure cloud. We present our attempt at a simple approach and it limitations as well as the challenges we encountered.

We also look at how well our solution scales, with respect to load handling, at different number of servers(horizontal scaling) and different hardware configurations(vertical scaling).

Preface

I had no idea what I went into when I started my master‘s degree in Informatics at the Norwegian University of Science and Technology 2013. I knew I had a passion for working with computers, the more the merrier.

My way towards a thesis has been a long and winding road with many unforeseen challenges. Not every day has been as good as the next, but thanks to a world class group of friends and family, we have pulled it trough.

I want to thank my supervisor Svein Erik Bratsberg for his persistent support when I have struggled to find a direction. Finding a topic for my thesis turned out to be way harder than I had ever imagined, and I probably would not have finished without him. I also want to thank my family for their continuous support, especially during my last year writing this thesis. My uncle Harrald Eri who helped me see my topic in a new light, his support has been invaluable.

Lastly I want to send a heartfelt thanks and goodbye to the student environment in Trondheim, especially Online and Realfagskjelleren, which has been a constant force for joy in my now 6 years of higher education.

Even Østvold

Trondheim, Mai 2014

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

As the cloud becomes a more and more accepted, more and more small businesses and individuals start to move their application environments to the cloud. Many of these will there applications on Open Sours technology and solutions they are familiar with from their previous endeavors.

The emergence of Big Data has brought with it changes in how data is stored. This new databases called NoSQL(Not only SQL) provides different ways of storing and accessing data. The interfaces of many of these NoSQL databases is built upon known protocols (like HTTP), and others define their own.

This diverse database designs pose a challenge when it comes to the cloud. With so many different solutions available it is not given that any given cloud supplier provides the database solution compatible with the client‘s needs. Most cloud suppliers offer some database technologies as a service (DaaS), and the applications may simply be rewritten to utilize these, but some might be more hesitant.

he cloud brings many benefits to businesses like capital cost savings and simplifies their operations. Maintenance of hardware, underlying operation system, and many security issues is handled by the cloud operator. On the down side the cloud is based upon shared resources and to ensure that one customer dos not hinder another the services provided is normally provided with restrictions.

For many of these businesses the solution might be to run their own database in the cloud. There exists many challenges in implementing a distributed database, but many of the NoSQL databases was invented after the emergence of the cloud and has taken them into account.

We will attempt to implement a NoSQL database, MongoDB in the Azure cloud with as little configuration as possible. After that we will do a series of performance tests to try to get a glimpse of an aspect hidden away by the Database as a Service (Daas), how well the database scales.

1.2 Problem Specification

The goal of this thesis is to implement a NoSQL database in the cloud. We will measure its scalability over different performance levels and see what benefits they offer.

Particular research questions we‘ll be exploring is:

- How can a NoSQL database be implemented in The Cloud.
- What challenges exist when implementing a sharded database in the cloud.
- How dos the implemented database scale.
- How dos the cloud performance impact the databases scalability.

1.3 Limitation

This research is limited to an implementation that provides database that uses shardng to store data on multiple servers and a measurement of the systems scalability. It does not go into an elastic(automatic increase and decrease of servers).

In addition, due to time restraints, redundant storage of data will not be implemented. In addition, has time made it necessary to limit the scalability tests to a write/load only test, as added write capabilities is one of the benefits of sharding compared to replication.

1.4 Thesis Outline

This thesis is divided into three main parts, Literature Review, Implementation, and Analysis.

Chapter 1, Introduction - an introduction to the thesis(this chapter). The chapter describes the scope, limitations, and structure of the thesis is presented.

Part 1, Literature review

Chapter 2, The Cloud - consists of a description as well as the main features of the cloud.

Chapter 3, Scalability - defines what we men with scalability and how we can measure it.

Chapter 4, Sharding - describes the horizontal partitioning scheme sharding.

Chapter 5, MongoDB - presents the NoSQL database we will be implementing in the cloud.

Part 2, Implementation

Chapter 6, Implementation - detail our process of implementing MongoDB in the Azure cloud, and the challenges we encountered.

Chapter 7, Results and Analysis - describes the measurements taken of the implemented system and there analysis.

Part 3, Analysis

Chapter 8, Discussion - is a discussion and summary of the experience, challenges and measurements obtained during the work on this thesis.

Chapter 9, Conclusion - we answer our research questions based on the discussion in chapter 8.

Chapter 2

The Cloud

A Cloud consists of many virtual servers on a distributed network that is used to pool and share resources amongst users. It has ushered in a paradigm change in the Information Technology(IT) sector due to its low cost of hardware. The main ideas behind the cloud has existed for many years sins the static terminals at the dawn of electronic computers enabled a way to manage the underlying resources for better utilization[29]. The modern day cloud providers (like Amazon, Google, and Microsoft) started to provide cloud services to better utilize their big server parks they had to handle peak load. Later the cloud provider roll escalated to a business operation in its own right with data centers being built specifically to facilitate it.

The United States National Institute of Standards and Technology(NIST) has defined Cloud Computing[24] as follows:

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models."

Two of the characteristics they mentioned is of special relevance to us here:

> "*Resource pooling.* The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model,.. . . "
>
> "*Rapid elasticity.* Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand."

The Cloud is also by many considered to be more than just than the technology that lies behind it. To fully utilize its potential developers must adopt a methodology that takes full use of the availability of hardware and the flexible prising schemes[32].

2.1 PaaS vs IaaS

The NIST defines to *Service Modles*[24] for the cloud, Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) where the former builds upon the later.

Infrastructure as a Service (IaaS) provides fundamental capabilities like visualized servers, storage, and network. The User is normal billed based upon the amount of resources they consumes. The user normal also has some degree of control over the configuration of the infrastructure like opening ports in a firewall.

Platform as a Service (PaaS) where the cloud provider in addition takes on the responsibility of the Operating System and related software and services so that the customer only need to focus on their application. The user might also be able to control some configurations relating to the provided platform.

The services offered as part of PaaS might greatly benefit the speed of application development and there prizing is often. What the different cloud providers offer as part of their PaaS varies .

2.2 Benefits and Disadvantages of Cloud Computing

One of the most direct benefits of cloud computing is its access to maintained servers at a low prize, something that is especially useful for small businesses that don't have a big IT department[23].

On the other hand, the cloud is often much less flexible than an on-premise solution[29]. Sins resources are shared the access to them might be hampered by other users, or limited (throttled) by the cloud provider to ensure a minimum quality of service to all the users[32].

Security is one a related benefit of cloud computing, especially with the larger providers. The application developers can focus on the security of their application, and the cloud provider secures the rest of the environment. This is a huge benefit as attackers would have to bypass the cloud provider, who not only has resources, but motivation to uphold a secure image in a competitive market[29, 32].

2.3 Windows Azure

Azure is Microsoft‘s cloud service[13]. Microsoft uses Azure to power many of their own solutions like *Ofice 360* and *Bing*[13]

Azure supports PaaS mainly through virtual Windows Server instances and IaaS with many predefined images ranging from basic Linux distributions, to preconfigured LAMP[1] configurations amongst others.

The PaaS solution is comprised of two main components; Web Roles and Worker Roles. The web roles are fitted with Internet Information Services and come configured to utilize the front end load balancer. The Worker Roles is fitted with .NET framework, though other environments can be installed post startup. A Azure solution can be comprised of several Web and/or Worker roles, where each role consists of several instances, all running the same software package.

Azure and its adhering services favor an indirect approach to communication[32]. They provide Queuing services(And is one of the few providers that have an At-Most-Once delivery guaranty with their Service Bus Queues[12]) and a message buss as their main form of communication and structuring of application(though no one is forced to play by these rules).

Azure provides these services and others for a small fee (based on the usage of the services like 1$ per 1mill requests). Besides services to related to the users solutions, Azure provides a federated SQL server[2], Table and Blob storage[3], and queuing servises[4] amongst others.

[1]LAMP stands for Linux Apache, MySQL and PHP and is a common solution stack for websites.
[2]Azure SQL Database - `http://msdn.microsoft.com/en-us/library/azure/ee336279.aspx`
[3]Storage - `http://msdn.microsoft.com/en-us/library/azure/gg433040.aspx`
[4]Queues - `http://msdn.microsoft.com/en-us/library/hh767287.aspx`

Chapter 3

Scalability

3.1 Defining scalability

André B. Bondi of AT&T Labs defined scalability as “The concept connotes the ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement”[8]. He further proceeds to give an initial taxonomy consisting of four types of scalability:

Load scalability describes a system that is capable of operating graceful different loads while making good use of available resources. Some of the factors that may hamper load scalability is scheduling of a shared resource, scheduling of a class of resources in a manner that increases its own usage, and inadequate exploitation of parallelism.

Space scalability refers to the growth of memory usage compared to the scale of the system. Many different approaches like space efficient algorithms and compression can help with space scalability, but the effects (like added CPU time of compression) might reduce other types of scalability like *load scalability*.

Space-time scalability regards the ability of a system functions gracefully when the number of items it handles increase by an order of magnitude. Space-time scalability may be related to both load scalability and space scalability in that the amount of items might stem from an increased load, and the presence of these objects may use more memory and affect data structures.

Structural scalability refers to the implementation or standards of the system and how they limit the number of item the system may handle. The prime example of structural scalability concerns the addressing of the items, for instance will a fixed addressing space put a limit on the systems scalability.

3.2 Quantifying scalability

Amdahl's law[6] is a model for parallel speedup constrained by it‘s the unavoidable serial paths under a fixed problem size. It assumes the serial portion of the algorithms remains the same regardless of the number of processors. The law consists of the speedup S, the number of processors N, and the non-parallelizable portion P.

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

Figure 3.1: Amdahl's law

Gustafson's law[21] addresses the shortcoming of Amdahl's law that it assumes the problem size for each processor remains constant regardless of their number.

The law consists of the number of processors P, the speedup S, and the non-parallelizable fraction α.

$$S(P) = P - \alpha \cdot (P - 1)$$

Figure 3.2: Gustafson's lawl

In summary, Amdahl's law describes how much faster a problem can be solved by adding processing power, while Gustafson's law implies that larger problem sizes can be solved in the same time by adding more processing power.

3.2.1 Universal Scalability Law

The Universal Scalability Law models the scalability of a system as a rational graph consisting of a second-degree polynomial. One of the biggest advantages of USL over earlier models is that it incorporates that non-linear scalable systems often has a point where the throughput starts to retrograde[19]. The model is not intended to predict the scalability of the system beyond the point where it starts to retrograde.

$$C_p(\sigma, \kappa) = \frac{p}{1+\sigma(p-1)+\kappa p(p-1)} \qquad 0 \leq \sigma, \kappa \leq 1$$

Figure 3.3: Universal Scalability Law[19]

The Universal Scalability Law incorporates three major effects[20]:

Concurrency represents the number of threads/users/processors used in the system. Here represented by p.

Contention is conflict over access to a shared resource(RAM, locks, I/O, etc.).It is here represented by the σ factor and corresponds to the serial portion described by Amdahl's law.

Coherency deals with the work needed to keep the state and/or data consistent across the system. it is here represented by κ. By excluding coherency (setting $\kappa = 0$) the USL reduces to Amdahl's law.

Gunter presents the following results to give easier insight into the peak scalability(p^*)[19]:

(a) $p^* \rightarrow 0$ as $\kappa \rightarrow \infty$

(b) $p^* \rightarrow \infty$ as $\kappa \rightarrow 0$

(c) $p^* \rightarrow \kappa^{-1/2}$ as $\sigma \rightarrow 0$

(d) $p^* \rightarrow 0$ as $\sigma \rightarrow 1$

The 'universal' part of the name is intended to convey that the model "can be applied to any computer architecture; from multi-core to multi-tier". The concurrency measurement p stems from either hardware ore software scalability:

Software scalability is characterized by a fixed hardware quantity and only the load(N) is increased.

Hardware scalability scales both the hardware and load proportionately. For example if we measure our system with 10 instances and 100 users, we would measure 15 instances with a load of 150 users.

Chapter 4

Sharding

The idea behind sharding is to split the data across multiple machines, and thus falls under horizontal scalability and ‘shared-nothing’ architecture. Sharding is mainly used to increase the volume of data supported by a database solution and increase the number of write queries by adding more hardware. This ability to utilize more hardware distributed over more servers enables the use of commodity hardware that both increases the availability of spare parts and reduces cost[11].

The proses of sharding increased in popularity with the emergence of NoSQL databases and the BigData[9] movement(though horizontal partitioning existed before). Many big companies has published database solutions that facilitates sharding like Amazon[17] and Google[9, 11, 28].

There exist different approaches to distribute data amongst shards:

Key-based partitioning is a scheme where the data itself is used to perform the partitioning. A challenge with this approach is to achieve a even distribution across the shards. Certain data types lend them self more easily for distribution, for instance monotonically increasing numbers (which can be partitioned easy with a modulo of the number of shards). A popular solution to achieve an even distribution with arbitrary data types is to use a consistent hash algorithm[22]. A disadvantage of the hashing approach is that it makes it practically impossible to associate different entities by their hashed data, and as a consequence of this, data might be needlessly separated.

Directory-based partitioning outsources the mapping between data and shard to an external lookup table. Some of the challenges associated with directory-based sharding is that the directories introduces additional points of failure, and the added communication cost.

Some of the challenges associated with sharding is that the increased number of servers increase the chance of any one of the failing. As stated data might be needlessly separated across servers. Depending on the database solution the sharding mechanism might complicate the application code.

Chapter 5

MongoDB

MongoDB is an open source document database built and structured to have high read and write throughput, easy scaling and redundancy [7]. It is developed by *10gen* who has made it available Open Source under the “GNU AGPL v3.0.” license (and the drivers under “Apache License v2.0”)[3]. The MongoDB server is interfaced through its custom binary protocol making custom drivers a necessity. This driver runs on top of TCP/IP using two streams, one for input, and one for output[1].

MongoDB organizes data in two level hierarchies. At the top level is the database, and each database consists of one or more collections. The schema less nature of MongoDB means that all data technically can be present in the same collections, but organization and the limitation on the number of indexes a collection can have, encourages the utilization of more collections.

5.1 Queries

MongoDB uses a JSON like query language including some predetermined and user provided(stored procedures) JavaScript methods and reserved set of operators[30]. The JSON formate provides ad hoc querying (not specified in advance)

```
var persons = db.persons.find({age: {$gte: 45}}).sort()
```

When a query returns, it returns a *‘cursor’* that points to the result on the server. The client can manipulate this cursor on the server(iterate over result(s), or retrieve parts of the result). To amend the above query to only return the fields for first and last name we would write:

```
var persons = db.persons.find({age: {$gte: 45}}, { firstName:
1, lastName: 1 }).sort()
```

to read in the actual content of the query result we would write:
```
persons.next()
```
When using MongoDB it is important to remember that it only enforces atomic operations on an individual document level. If a manipulations of several documents is to be consistent, that responsibility falls to the client.

5.2 Stored procedures

MongoDB lets you create JavaScript functions and store them on the server. They will be stored at the server directly connected to the client. Even in the case that the server is a mongos instance, the procedure will not be propagated trough the sharded system.

To add a stored prosecure named ‘add’, insert it as a JSON document to the ‘js’ document in the ‘systems’ collection.
```
db.system.js.save({_id:"add", value:function(x, y)
                     { return x + y; }});
```
The *save* command creates a document if it does not exists. If the document do exist, the JSON will be amended to it. To execute a stored procedure we use the ‘eval’ function:
```
db.eval(‶add(21, 21)″);
```

5.3 Index support

MongoDB implements indexes as B-trees, and any collection can support up-to 64 indexes. All documents automatically have an index on its *‘_ID’* field which is called the primary index. Each collection can hold up to 63 other secondary indexes and they can be of varying types like compound, multi-keyed, text and hashed, amongst others.

To add a text index to the fields ‘revision.text’ and ‘revision.comment’ we would issue the command:
```
db.collection.ensureIndex({ ‶revision.text″:  "text",
                             ‶revision.comment″:  "text" })
```

5.4 Consistent architecture

MongoDB supports many different configurations; single server, replicated servers, sharded cluster, and even sharded cluster with varying number of replicas per shard[10]. The basic interface provided by a single MongoDB server is transparently and consistently available throughout all these configurations. Replication(Replica sets) and horizontal partitioning(sharding) may be present, and even change without the client needing to be any the wiser. These features has their own interfaces(*rs* for Replica Sets, and *sh* for Shards) that is manipulated as any other MongoDB document, trough JSON queries.

An example of a replica set query to add another server to the set:
`rs.add( { "_id":1, "host":"localhost:27017", "priority":0 } )`
And an example of a query adding another server to a sharded cluster:
`sh.addShard( "localhost:27017" )`

5.5 Built in Sharding support

MongoDB was from the start designed to support sharding, and support was first added in version 1.6[7]. The data storage of the sharding system is stored in regular MongoDB instances. These can all be used as regular databases, but will only show the designated part of the entire dataset. To interface with the entire database MongoDB employs a router system called `mongos`. These routers are stateless and stores the clusters metadata in three normal databases set in configuration server mode. In configuration mode all servers is kept consistent through the use of a Two-Phase commit.

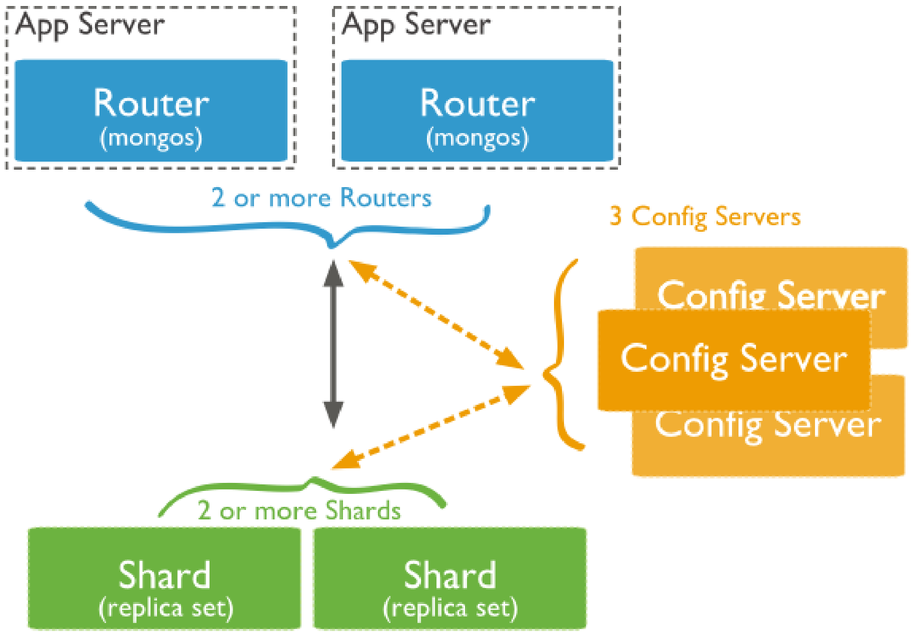

Figure 5.1: Diagram of MongoDB sharded Cluster (taken from MongoDB documentation [1])

[1] Diagram of MongoDB sharded Cluster. Accessed on 5/3-2014
http://docs.mongodb.org/manual/core/sharding-introduction/

MongoDB distributes data through a range-based approach, see fig 5.2. All collections that are to be sharded must specify a field that is used to partition the data. The standard approach in MongoDB is to use the lexical value of the field to create an ordered range. Then values are split into *chunks* by selecting boundary values. For instance chunk 1 consist of all documents starting with ‘a’ trough ‘f’, and chunk 2 consists of ‘g’ trough ‘m’.

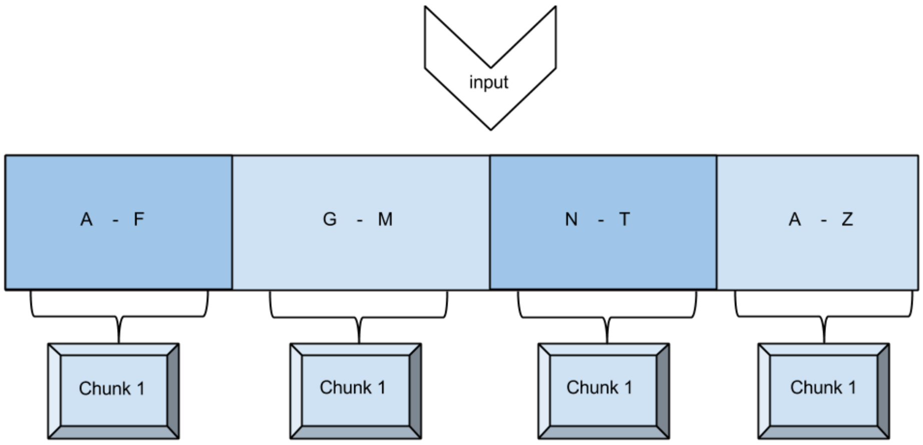

Figure 5.2: Illustration of range based partitioning

MongoDB also provides the ability to hash the shard key before it is placed into the range in a process called *auto-sharding*. The auto-sharding approach has a significantly higher chance of spreading the data evenly across the chunks reducing administration (hence the name auto-sharding). The downside of the auto-sharding approach is that the application loses control over which document is likely to be in the same chunk/server as another.

5.6 High availability

MongoDB suports high availability trough what it calls *replica sets*. A replica set is a group of normal mongod databases that cooperate on holding redundant copies of the database. Each replica set elects one leader who is responsible for coordinating writing of data, while all the members can facilitate reading[10]. In case of node failure, MongoDB handles the fail over automatically. In a sharded cluster the number of replicas on each shard is independent.

When accessing a replica set the client the consistency demands by providing a write concern named *MAJORITY* which requires a success from a quorum of at least half the replicas before it succeeds.

5.7 Existing cloud solutions

There exists today several vendors offering MongoDB as a service. 10gen has a list of their 'cloud partners'[5] who provides MongoDB services through both public and private clouds, and most of these come with sharding capabilities as one of the options. Of special interest in our work on this thesis is the white paper generally outlining a strategy for implementing MongoDB on Amazon Web Services[31].

10gen. has also released a tutorial on how to implement MongoDB on Azure[2], this solution however only deals with replica sets and not shading. Their solution is implemented using C#. Our knowledge of C# is not the best, but we have managed to extract useful information.

Chapter 6

Implementation

To implement our experiment we have used Eclipse[18] workbench for Java with an Azure plug-in[25] and SDK[26] from Microsoft Open Technologies[27]. The plug-in facilitates easy configuration of, and deployment to, the Azure cloud.

We chose to use MongoDB for this project mainly because it is a very popular database. The website `db-engines.com` has a ranking[16] over the most popular databases measured on a variety of factors like Google trends, mention in job offers, and relevance in social networks amongst others. MongoDB comes in on a 5th place, only beaten by established SQL databases like Oracle, the closest NoSQL database is Casandra on 9th place.

6.1 MongoDB on Azure

The implementation of the sharded MongoDB solution is divided into three parts; the databases, the routers and the configuration databases. These parts is the result of a direct mapping from MongoDB‘s shard structure as seen in fig 7.4. Each of these parts is designated their own *role* in Azure as shown in fig 6.1. This enables each part to be scaled independently. Each role consists of the MongoDB binaries and a startup script that executes the binaries and connects them together.

To enable the three roles to communicate with each other ports where opened in the internal Azure firewall. We opened the ports by defining an internal endpoint on each role. With internal endpoints the firewall is opened and is accessible from all the instances in the deployment by default. There exists options for defining more elaborate rules for the firewall, but that is not necessary for our solution. The endpoint are defined in Azures ‘ServiceConfiguration’ file.

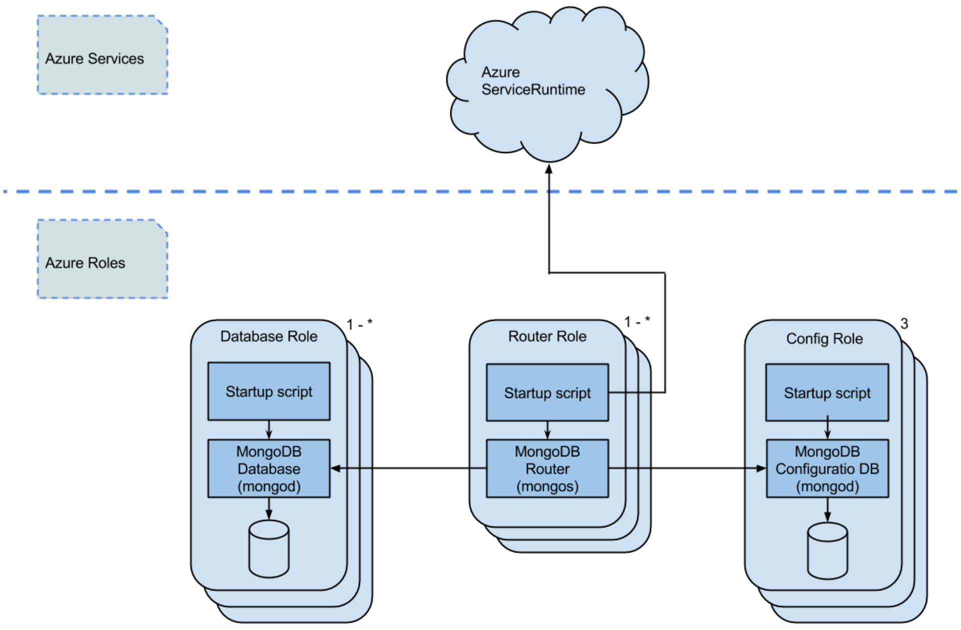

Figure 6.1: MongoDB shard components mapped onto Azure Roles

The main challenge in the implementation was to make the different instances talk to each other. One of the main characteristics of a sharded database is that the data tis separated over different servers, and as a consequence requests need to be routed to a particular server/instance. This is fundamentally different to how communication is intended to happen inside Azure(see 2.3). Our solution to this challenge is to use the *Azure Service Runtime API*(ASA). In contrast to azure storage services who utilize a RESTfull[14] api(RESTfull services is built upon the HTTP protocol which is i most common tools), the ASR don't, and need to be accessed through the Azure SDK. Our solution let one look up the IP address and port of instance endpoints, enabling direct communication. One result of this approach is that the our solution is strongly coupled, and would be very hard to scale elastically(Scale inn and out during runtime, often in response to demand[32]).

One other simple but noteworthy 'obstacle' when using Azure for the first time is that it is capped at 20 cores(not instances, but cores. Medium instances consists of 2 cores...). This is easily fixed by contacting the Azure Support Center. Introduced as a safety feature to avoid runaway cost to new customs, this feature can also be annoying until one figure out how to solve it.

6.1.1 Configuration server

The simplest part of the sharding solution is the configuration servers. These servers are normal MongoDB servers(the standard binaries), but run in configuration mode(run with a '–configsvr' argument). All the configuration servers is set to listen for incoming connection on port 27051. The main challenge posed by these servers is that they must be identifies to the *routers* directly by IP.

Besides this there are no significant challenges. All requests don to the configuration servers is consistently replicated across all the servers, so no special care to the individual instances is needed.

6.1.2 Router server

The sharding routers act as the interface trough witch the sharded cluster is accessed. As such it is in need of information on the entire sharded database. This information is present in *configuration servers*. These servers must be identified at the application startup, and if changes to the makeup of these configuration servers where to occur at a later point, all parts of the sharding solution(config, router, and database) has to be restarted[4]. The routers are configured to listen to port 27019. To ensure that the configuration servers is up and running when we start the router the startup script enters a wait cycle where it repeatedly tries to connect to the configuration database as a normal database(see fig 6.2). One this connection is established we know the server is up and can launch the router application.

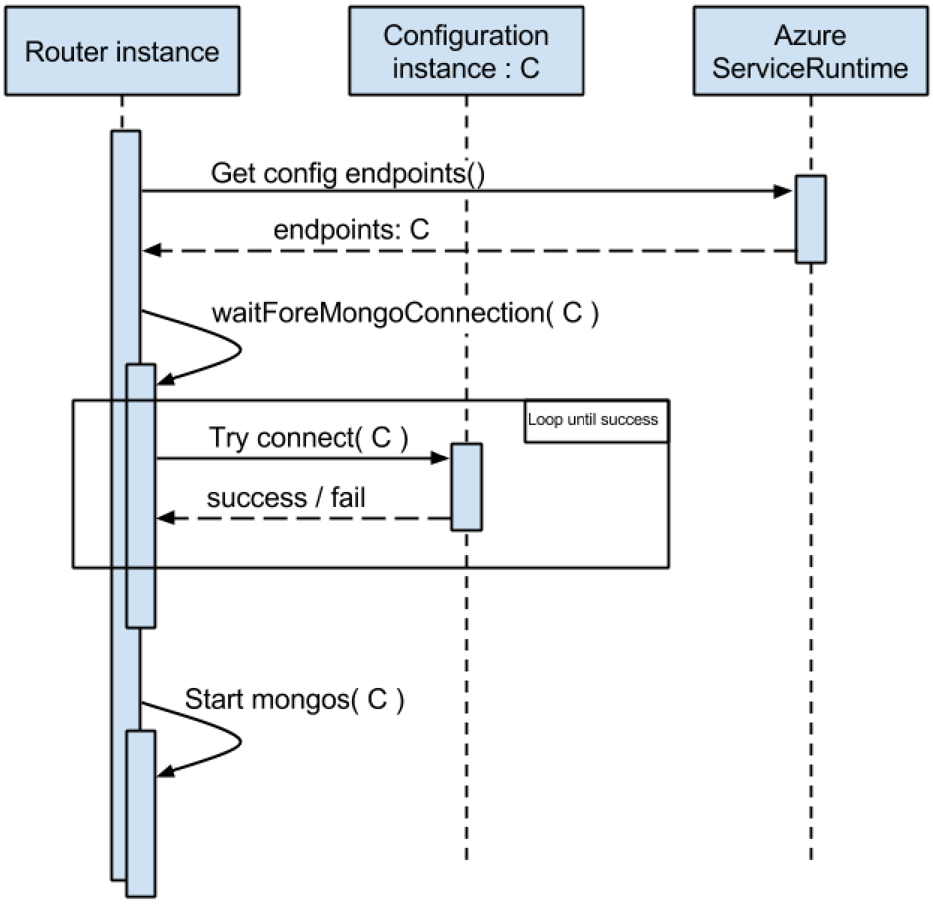

Figure 6.2: Router startup script

6.1.3 Database server

The database server is quite like the *Configuration server* in that it operates without knowledge of the rest of the sharding solution. Once a database server is up and running it is up to the *router server* to include it in the greater database by including it in the configuration database a directing traffic to it. In our solution we have opted to let the instance hosting the database server is responsible for contacting the router and adding itself to the greater database. This is because this is the path of least resistance in that the only external information needed is the address of one router. This not only makes it easy to start the cluster, but also adding extra shards post launch. The databases listen to the default port of 27017.

To ensure the router is up and running before the database attempts to connect to it, the database server employs the same wait cycle scheme as the router with the one change that it looks up an address of a router instead of a configuration server.

Chapter 7

Results and Analysis

7.1 Measuring performance

To measure the performance of our database solution we will be using a small Java program based upon the Executor framework [1] (details explained in section 7.2). In addition we will be using the tools *mongostat* and *Windows Performance Counters* to look at the state of the individual instances.

Mongostat is an utility program that is part of the MongoDB distribution[15]. It provides statistics at regular intervals about vitalities of both databases and routers(all parts of the sharding cluster) like number of reads, writes, network, and lock status to mention a few. It dos however not include any information on its disc usage. The output is in a CSV like format that is stored in a file and collected for analyzed after each test is performed.

Performance Counters is a part of the Microsoft Windows operating systems, both personal and server[15]. It gives measurement on a vast variety of metrics and can be focused on the operating system and/or other programs. The Azure platform provides a 'Diagnostics' module that facilitates collection of performance counters and storage in Azures Table Storage. Unfortunately we have not been able to get this feature to work. Instead we stored the results to a file and collected it later, the same as with *Mongostat*.

The last form of measurements we are collecting comes from the application performing the tests. For the batch inserting of data(the loading of the database) we collect the number of inserted revisions and the accumulated size of these collections.

[1]Executors (The Java^{TM} Tutorials). Accessed 5/3-2014.
`http://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html`

7.1.1 Selection of dataset

To populate the database we used a dump of the English Wikipedia database with revision history. These datasets is freely available at download.wikipedia.com. Our main reason for choosing this dataset is that it is freely available, its size, and that it has some internal relations. The content itself is not of relevance, but the English version was chosen based on its popularity in research.

The choice to use the Wikipedia dataset was to have real world data of that contains a sizable amount of data and a variety of relations [2].

The dataset is available in different versions like 'all pages with complete edit history' and 'All pages, current versions only' amongst other smaller extracts like 'Wiki external URL link records'. The larges datasets is broken down into several independent files or segments. We chose to only use the first segment of the pages with revision history. It contains the pages 'A' to 'Blues' and is 19GB zipped and extracted. A summary of the dataset can be seen in fig 7.1.

An example of an entry from the "AccessibleComputing" page can be seen in fig 7.2. The 'timestamp', 'id', and 'contributor' are suitable for retrieval, especially the id being unique. The 'text' and 'comment' fields are suitable for text search.

Distribution of revisions per page					
min	1Q	Median	Mean	3Q	Max
0.0	32.0	230.0	904.8	903.0	16827.0

Distribution of revisions size in bytes					
min	1Q	Median	Mean	3Q	Max
281	1620	7432	14550	20400	157100

Figure 7.1: Distribution of revisions on first 2157 English Wikipedia pages

7.2 Performing tests

The tests will be performed by a Java program that is present on every Router instance. The tests are started manually after the instances have been initialized and the test data automatically downloaded. The test program communicates with the MongoDB router through the MongoDB API for Java by connecting to the local port 27019. Each instance of the test program collaborate to insert the full dataset by inserting one portion each, achieved by a modulo test at the page level of the dataset.

Threads are used to simulate users, with one thread counting as one user. The life cycle of the thread is to fetch a task from a queue, perform that task, register statistics about that task, an repeating until all the tasks is depleted(See fig 7.3).

[2]An interesting possibility not explored in this thesis is the impact of relations present in the data, that is not present in the model. The text field of the Wikipedia dataset contains a potential for these.

```
{
    "revision": {
        "timestamp": "2001-01-21T02:12:21Z",
        "id": 233192,
        "model": "wikitext",
        "text": "<Page text goes here>",
        "contributor": {
            "id": 99,
            "username": "RoseParks"
        },
        "sha1": "8kul9tlwjm9oxgvqzbwuegt9b2830vw",
        "format": "text/x-wiki",
        "comment": "<User comment goes here>"
    }
}
```

Figure 7.2: EnWikis JSON example

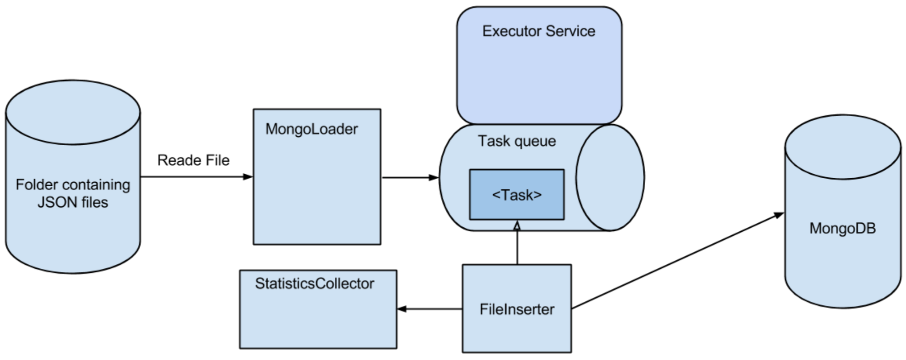

Figure 7.3: Lode Generator domain diagram

An important part of the MongoDB API when i comes to writing data to the database is the *Write Concern*. The weakest write concern is *UNACKNOWLEDGED* that is an asynchronous operation that succeeds one the request has been written to the senders socket. A write concern of *ACKNOWLEDGED* considers the write request a success one the server has revived the request. A stronger concern is *JOURNALED*, witch succeeds one the request is successfully written to the journal, adding fault tolerance. The strongest write concern is *FSYNCED*, which only returns one the write operation has been written to disc. These configurations affect the testing program quite a bit as they affect the behavior of the statistics collected. With unacknowledged, the tests would be useless, sins the only thing we would be measuring was the routers outgoing network connection. Acknowledged would work better, but have the result of sewing the data with a high throughput at the start that tapers off as the database gets overloaded and refuses more incoming writes. Clearly what we want is fsynced. This not only gives the most accurate statistics, but also enables a helps in connecting the statistics to other metrics measured at the database. We will be using fsynced on all the write operations.

The statistics we will be collecting is the number of revisions successfully inserted, and the size of the revision as an extra validity check. We are only interested in the aggregated values, and so all the threads report to the same statistics collector. Each 5th second a sample is taken of the statistics and reported. The five seconds is chosen as a good trade of between performance and fidelity. These statistics is then written to a file fore later collection and analysis.

We will run our tests against a database consisting of 1, 2, 5, 10 and 15 shards. The limit of 15 instances was chosen on the basis of cost limitations. Each large instance in Azure consists of 4 cores, and with 15 shards makes it 60 cores just for the databases. In addition we will run the tests against a database that is not sharded for a comparison. Each of these tests will be performed in isolation, and each will be performed against an empty database. These tests will be repeated in the context of small, medium, and large role sizes and there results compared to see how the added performance(vertical scalability) compliments the horizontal scalability.

7.3 Results

Our tests of the scaling of MongoDB‘s load capabilities start with some foundational data about the system. In figure 7.1 we present the basic load throughput measured by our test system on the different instance sizes as a baseline.

Size	**Average # of revisions inserted per 5 seconds**
Small	1303
Medium	1998
Large	656.5

Table 7.1: Results of load test on normal (unsharded) databases

7.3.1 Bulk data insertion

In our first attempt to insert data we had represented each page from the English Wikipedia as a collection containing its revisions. This went well until we attempted to use 10 shards(and 100 users on 10 routers). With this workload the solution broke down after just 15 minutes giving of the following error:

```
"exception:  socket exception [CONNECT_ERROR] ..."
"exception:...transport error:...{ splitChunk:...}"
```

The cause of these errors where traced to the *configuration servers* being overloaded. To test whether this was the cause of the problems a similar run was performed, but where all the collections where added before the tests(without data).These changes made the tests run for 45min before the errors returned. Proceeding from these findings we made the change that all the pages where to be inserted into the same collection (this is safe because all pages and there revisions have unique id‘s used both to identify the documents and as a shard key. This solved the problem.

7.3.2 Measurement overview

In figure 7.4 we have compared the results of the load tests performed on the three instance sizes with 1, 2, 3, 5, 10, 15 and 20. The medium instances will end at 15, and the large instances will end at 10. The reason for the deviation from the original plan of reaching 15 shards for the large instances was that we reached our policy limit a little sooner than anticipated(limit on number of cores that can be active).

The first thing we notice is that there is a throughput gain in the favor of the larges instances(vertical scalability). This is as expected as the larger instances all have better disk I/O speeds.

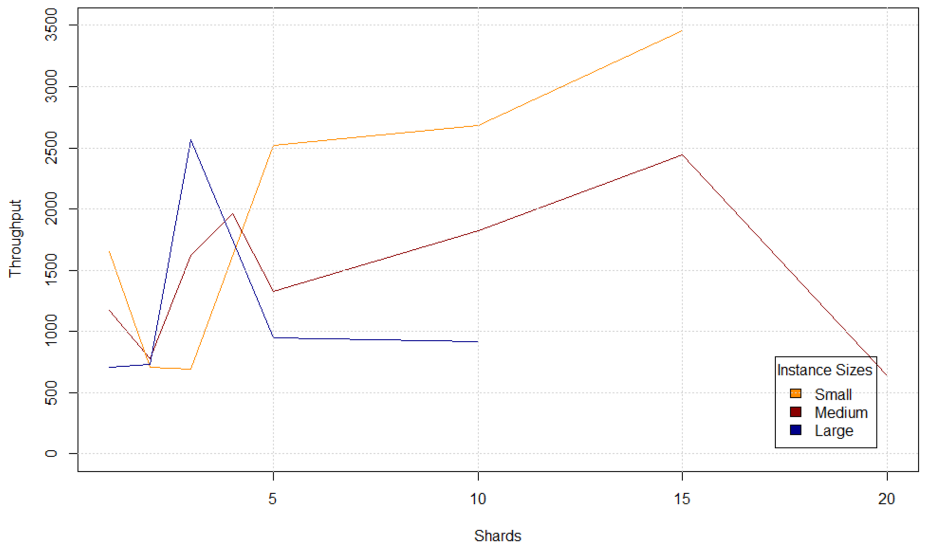

Figure 7.4: Comparison of throughput to shards on small instances

Suspecting the load balancer

The second thing we notice is that our solution is not scaling as well horizontal as we had anticipated. At two shards we see a marked drop in throughput in all the instance sizes. This drop is somewhat expected as this is the first configuration where the system needs to split data between shards. During a closer examination we noticed that the system was performing some chunk migration in the background. To determine to what extent this affected the system we performed a test where we first measured the throughput with the load balancer on, and then again with the load balancer of, to determine what effect this had on the overall throughput. The results are shown in fig 7.2 and shows no significant effect. This is to be expected as we are using a hash based approach to split the data amongst the shards, which helps to uniformly distribute the data.

Balancer	**Max average throughput**
On	1763.5
Off	1736.5

Table 7.2: Results of turning the load balancer off on small instances

Analyzing mongostat logs

To try to get a better understanding of what was going on we consulted the *mongostats* logs from the tests to see if the could shed some insight(See fig 7.5). We see that there is some activity going on, but not much. More to the point, there is plenty of resources available. Of special importance is the 'locked db' fields as MongoDB aquires an exclusive write lock to the entire database(local shard) for each write operation. The network usage is neither likely to be the cause of our trouble. In summary, the mongostat results backs up the pore performance, but offer no help in finding the cause.

Turning to the Performance Counters

The performance counters held nothing of interest either. A little amount of data was revived over the network, and a little bit of data was written to disc. This is not surprising. The Performance Counters collect general data and has no insight into the applications that is a port of this test, only the operating system hosting them. We would expect the performance counters to be more of use in a situation where the instances where being overloaded

For a more detailed overview of the test results see appendix A.1.

1 shard:

```
insert   mapped   faults   "locked db"      netIn    netOut   conn
61       544m     1633g    em:5.1%          2m       7k       22
183      544m     3713g    em:7.4%          6m       18k      22
140      1.03g    2716g    em:38.2%         4m       14k      22
163      1.03g    3146g    em:21.0%         5m       16k      22
```

2 shard:

```
insert    mapped   vsize   "locked db"  NetIn  netOut   conn
0         468m     47m     em:0.4%      15k      1k     28
11        468m     51m     em:0.9%      179k     2k     28
7         469m     51m     em:0.7%      73k      2k     29
```

10 shard:

```
insert   mapped   faults   "locked db"      netIn    netOut   conn
2        160m     532m      em:0.1%         11k      2k         92
1        160m     543m      em:0.0%         6k       2k        103
1        160m     553m      em:0.1%         10k      1k        113
3        160m     553m      em:0.3%         15k      1k        113
```

Figure 7.5: Selected excerpts of *mongostat* log from a small instance test

7.3.3 Summary

Our solution is running in the cloud and accepting data, and it is possible to increase the throughput beyond threshold of what one single database could muster. Lifting our gace a little, it is obvious that our database dos not scale as we had predicted. MongoDB should be able to scale close to linearly, and that is far from what unfolded during our tests.

We see an acute scaling problem with our database solution on small instances. Our solution begins to retrograde once we pas 10 shards. This is to us the biggest surprise during this thesis.

The medium instances fear much better than the small. They have a higher throughput instance by instance with the small instances. At the end of the scalability test there were no signs of them starting to retrograde.

The large instances achieved a high throughput at 3 shards, but becoming the poorest performing instance size after that. Here we are at a complete loss. We don‘t know what, but something, somewhere is very wrong.

We feel certain that there is at least one factor limiting our solution, but we have been unable to determine what that factor is, or even where that factor lies. We have rerun several of the tests, changed out parameters, but achieved no meaningful deviance. At the end of it all, the only thing we feel certain about is that the measurements we have provided truthfully represents the system we have been testing.

7.4 Implementing MongoDB on Azure

Trough out this thesis we have found Windows Azure to be a fine tool to work width. Its decomposition of applications into a collection of roles worked well with our solution.

The emulator that comes with Azure was a real help in the beginning to test out the layout. We did however run into problems when we discovered that the worker roles was limited to only one instance per role(Web roles can sport multiple instances). Even though our application launches as a stateless application, and can verify its majority of features trough the emulator, there were times when we were working on bugs the ability to have more instances would have come in handy.

The tools provided by Microsoft, and more relevant to Java developers, Microsoft Open Technologies are truly great, and simplify the development process. We do not think we could have achieved the technical aspects of this thesis if it had not been for these tools.

Chapter 8

Conclusions

MongoDB is a database that from the start was designed to be used in the cloud. That sad, we must admit that we underestimated the challenges of implementing the database in, and take advantage of, the cloud. We have encountered several problems, especially when it comes to predicting the amount of time required to implement and test our solution.

8.1 Research questions

The amount of unexplained results form our implementation has lead us to the conclusion that we can‘t conclude with any measure of certainty regarding our implementation.

How can MongoDB be implemented in the Cloud:
To get a simple, one instance, version of MongoDB to running in the cloud is pretty straight forward. Azure‘s separation into roles fits nicely with the components of MongoDB sharded cluster. Our solution was a simple deployment that utilized just a few of MongoDB‘s features as well as Azure‘s. Even though we managed to get a sharded MongoDB ro run, and accept data in the cloud, we are not able to call it a success.

Our recommendation to those who would implement a sharded version of MongoDB in the cloud is to focus on a configuration server to manage your deployment instead of trying to jerry rig into the cloud vendor‘s system. and we would recommend building a separate system to manage the solution. Focus on getting for 1-3 shards before trying to benchmark the system.

What challenges exists when implementing a sharded database in the cloud:
Windows Azure don‘t sport a straight forward way of getting a hold on the endpoints of other instances, and we have come to the realization that that is for the most part a good thing. Our simple sharding solution managed to use the information to connect its different parts together, but with the inconclusive tests we performed it is difficult to say if it was a suitable solution.

How dos the implemented database scale:
Though we had big problems with the performance of our solution, we were able to achieve better throughput by both scaling vertically, and scaling horizontal. The fact that we achieved this we credit to the ready-made MongoDB.

We had some problems scaling the small instances past 10 shards and only managed to get a little more than twice the throughput out of the sharded solution compared to a ‘normal’ database. We never reached a peak throughput of the bigger instances, but also these had a far less than linear scalability.

This is not what we had expected as all data operations in a sharded cluster is supposed to be confined to the individual shards. We suspect the cause of this to lie in either the configuration database and/or our load tests. We have not been able to determine any closer what is causing this effect.

How dos the cloud performance impact the scalability:
When it comes to the load scalability of our implementation we see a, not so surprisingly, dependence on the underlying disc system. The bigger the instance, the more load it can handle. Due to all the unanswered questions about the performance of our system we are hesitant to draw a conclusion on the horizontal scalability aspects related to it.

8.2 Future work

We would like to get to the bottom of what is causing the problems with the reduced efficiency as more shards are added. If the way we performed our load tests where to be found the reason behind the problems, it might shed light on how not to interact with a sharded MongoDB cluster.

A variation of our solution that would have been interesting to work on is to have one MongoDB modules per core instead of per instance. We belie that to add more capabilities to each instance would help better exploit the resources, resulting in more value for investment.

This thesis has covered a sharded cluster without replication sets, and performed a simple load test on it. An interesting topic would be to see how the addition of replication sets(high availability) would impact this scalability. The official MongoDB documentation highly emphasizes the need to use replica sets.

Our short experiment with combining shards with redundant storage has shown us the challenges of coordinating different components of a system. The inherent state full nature of the shards means that in order to achieve an elastic system(Dynamic scale out/inn) coordination is needed. We envision that such control system not necessarily need to be limited to a database system, but any solution that combines multiple modules in one instance.

A topic that fall outside the scope of this thesis, but that have cough our interest is a of comparison the scalability with other systems. In particular we are interested in how our sharded MongoDB(or another NoSQL database) would compare to a sharded SQL database at different normalization levels. With all data in one table and no abstract methods.

Bibliography

[1] 10gen Inc. Mongodb wire protocol — mongodb meta driver manual 0.1-dev. `http://docs.mongodb.org/meta-driver/latest/legacy/mongodb-wire-protocol/`. Sins the current documentation " is currently in draft status and has not been approved or finalized", the legacy information was used.

[2] 10gen Inc. Windows azure. `http://docs.mongodb.org/ecosystem/platforms/windows-azure/`, 2014-02-19.

[3] 10gen Inc. Mongodb licensing. `http://www.mongodb.org/about/licensing/`, 2014-02-20.

[4] 10gen Inc. Config server availability. `http://docs.mongodb.org/manual/core/sharded-cluster-config-servers/#config-server-availability`, 2014-03-17.

[5] 10gen Inc. Get mongodb in the cloud | mongodb. `http://www.mongodb.com/partners/cloud`, 2014-03-19.

[6] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. URL `http://doi.acm.org/10.1145/1465482.1465560`.

[7] Kyle Banker. *MongoDB in action*. Manning, Shelter Island, NY, 2012. ISBN 9781935182870.

[8] André B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 195–203, New York, NY, USA, 2000. ACM. ISBN 1-58113-195-X. doi: 10.1145/350391.350432. URL `http://doi.acm.org/10.1145/350391.350432`.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach,

Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267308.1267323.

[10] Kristina Chodorow. *MongoDB, the definitive guide*. O'Reilly Media, Sebastopol, Calif, 2013. ISBN 978-1-449-34468-9.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay, Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, , and Dale Woodford. Spanner: Google's globally-distributed database. pages 251–264, 2012.

[12] Microsoft Corporation. Azure queues and service bus queues. http://msdn.microsoft.com/en-us/library/hh767287.aspx, 2014-02-06.

[13] Microsoft Corporation. Microsoft data centers | cloud resources. http://www.globalfoundationservices.com/cloud-resources.aspx, 2014-02-06.

[14] Microsoft Corporation. Storage services rest api reference. http://msdn.microsoft.com/en-us/library/azure/dd179355.aspx, 2014-03-17.

[15] Microsoft Corporation. Performance counters. http://msdn.microsoft.com/en-us/library/windows/desktop/aa373083(v=vs.85).aspx, 2014-03-17.

[16] Microsoft Corporation. Db-engines ranking - popularity ranking of database management systems. http://db-engines.com/en/ranking, 2014-03-17.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294281. URL http://doi.acm.org/10.1145/1323293.1294281.

[18] Eclipse Foundation. Eclipse project. http://www.eclipse.org, 2014-02-03.

[19] N. J. Gunther. A General Theory of Computational Scalability Based on Rational Functions. *ArXiv e-prints*, August 2008.

[20] Neil Gunther. *Guerrilla capacity planning a tactical approach to planning for highly scalable applications and services*. Springer, Berlin, 2007. ISBN 978-3540261384.

[21] John L. Gustafson. Multiprocessor performance measurement and evaluation. chapter Reevaluating Amdahl's Law, pages 92–93. IEEE Computer Society Press, Los

Alamitos, CA, USA, 1995. ISBN 0-8186-6522-X. URL http://dl.acm.org/citation.cfm?id=201945.201962.

[22] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM. ISBN 0-89791-888-6. doi: 10.1145/258533.258660. URL http://doi.acm.org/10.1145/258533.258660.

[23] Zaigham Mahmood. *Cloud computing methods and practical approaches*. Springer, London New York, 2013. ISBN 978-1447151067.

[24] Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011.

[25] Inc. Microsoft Open Technologies. Azure plugin for eclipse with java. http://msdn.microsoft.com/en-us/library/azure/hh694271.aspx, 2014-02-03.

[26] Inc. Microsoft Open Technologies. Windows azure sdk for java. http://msopentech.com/opentech-projects/windows-azure-sdk-for-java/, 2014-02-03.

[27] Inc. Microsoft Open Technologies. Microsoft open technologies, inc. http://msopentech.com/about/, 2014-02-03.

[28] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littlefield, and Phoenix Tong. F1 - the fault-tolerant distributed rdbms supporting google's ad business. In *SIGMOD*, 2012. Talk given at SIGMOD 2012.

[29] Barrie Sosinsky. *Cloud computing bible*. Wiley John Wiley distributor, Indianapolis, IN Chichester, 2011. ISBN 978-0470903568.

[30] Ciprian-Octavian Truica, Alexandru Boicea, and Ionut Trifan. Crud operations in mongodb. *Proceedings of the 2013 International Conference on Advanced Computer Science and Electronics Information*, 2013. doi: 10.2991/icacsei.2013.88. URL http://dx.doi.org/10.2991/icacsei.2013.88.

[31] Miles Ward. Mongodb nosql database on aws. http://info.mongodb.com/rs/mongodb/images/AWS_NoSQL_MongoDB.pdf, 2014-03-19.

[32] Bill Wilder. *Cloud architecture patterns*. O'Reilly Media, Sebastopol, CA, 2012. ISBN 978-1449319779.

Appendices

Appendix A

Measurements

All measurements taken with a 5 seconds sample rate.

A.1 Load results

Small instances

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0	906	1361	1303	1788	3598

Table A.1: Insertions into 1 small regular MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.0	951.2	1265.0	1172.0	1494.0	2154.0

Table A.2: Insertions into 1 small sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.0	131.0	398.0	382.6	594.0	929.0
2	0.0	110.0	427.0	391.5	629.5	918.0

Table A.3: Insertions into 2 small sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.00	92.75	616.00	531.50	832.50	1624.00
2	0.0	90.0	616.0	540.2	829.0	1968.0
3	0.00	77.75	624.00	549.60	864.80	1526.00

Table A.4: Insertions into 3 small sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.00	49.75	291.00	247.10	399.00	551.00
2	0.0	60.0	340.0	268.4	417.0	640.0
3	0.0	61.0	348.0	268.5	410.5	724.0
4	0.0	50.5	330.0	270.2	422.5	673.0
5	0.0	50.5	330.0	270.2	422.5	673.0

Table A.5: Insertions into 5 small sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.0	51.0	119.0	177.9	320.5	537.0
2	0.0	52.0	123.0	178.8	336.0	543.0
3	0.0	50.0	128.5	181.0	315.0	582.0
4	0.0	50.0	128.0	179.1	317.0	519.0
5	0.00	55.75	130.00	185.60	347.20	532.00
6	0.0	52.0	134.5	184.3	338.0	545.0
7	0.0	53.0	125.0	183.9	326.0	544.0
8	0.0	52.0	133.0	188.5	353.0	539.0
9	0.0	53.0	128.0	181.6	338.0	528.0
10	0.0	50.5	127.0	183.5	344.5	524.0

Table A.6: Insertions into 10 small sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.00	7.25	57.50	65.26	102.80	302.00
2	0.00	2.50	56.00	73.55	122.00	343.00
3	0.00	8.00	48.00	59.09	84.00	305.00
4	0.00	7.00	49.00	56.39	87.00	202.00
5	0.00	9.00	47.50	51.91	82.00	289.00
6	0.00	9.50	45.00	60.99	95.50	242.00
7	0.00	8.00	32.00	57.46	84.00	463.00
8	0.00	11.00	45.00	51.15	72.00	400.00
9	0.00	10.00	43.00	66.10	95.75	731.00
10	0.00	9.00	71.50	72.14	109.50	411.00
11	0.0	7.0	55.0	94.1	122.5	985.0
12	0.0	31.0	118.0	136.4	194.5	1441.0
13	0.0	17.0	64.0	101.2	135.0	1268.0
14	0.00	12.25	58.50	87.54	105.00	640.00
15	0.0	6.0	35.0	65.2	65.0	625.0

Table A.7: Insertions into 15 small sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.00	0.00	33.00	31.59	53.00	237.00
2	0.00	1.00	35.00	32.43	55.00	109.00
3	0.00	1.00	35.00	34.06	58.00	213.00
4	0.00	1.00	35.00	33.41	55.00	207.00
5	0.00	1.00	35.00	34.29	56.00	305.00
6	0.00	0.00	31.00	30.35	52.00	203.00
7	0.0	0.0	31.0	30.3	51.0	185.0
8	0.00	0.00	32.00	32.51	55.00	252.00
9	0.00	0.00	32.00	31.86	53.00	244.00
10	0.00	0.00	32.00	30.53	53.00	107.00
11	0.00	0.00	31.00	30.42	52.00	118.00
12	0.00	1.00	32.00	35.03	55.00	270.00
13	0.0	1.0	34.0	32.3	53.0	165.0
14	0.00	0.00	33.00	31.68	54.00	261.00
15	0.00	0.00	32.00	33.04	54.00	301.00
16	0.0	0.0	32.0	31.3	54.0	260.0
17	0.00	0.00	32.00	31.29	53.75	138.00
18	0.00	0.00	29.00	29.78	51.00	165.00
19	0.00	1.00	34.00	33.88	55.00	301.00
20	0.00	0.00	33.00	32.31	55.00	179.00

Table A.8: Insertions into 20 small sharded MongoDB. 5 sec sample frequency.

Medium instances

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0	1342	2194	1998	2729	5439

Table A.9: Insertions into 1 medium regular MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0	1159	1739	1649	2248	4170

Table A.10: Insertions into 1 medium sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.0	142.8	428.0	346.7	488.0	700.0
2	0.0	158.5	426.0	358.4	510.0	719.0

Table A.11: Insertions into 2 medium sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.0	70.5	304.0	232.7	353.0	466.0
2	0.0	68.5	293.0	220.5	340.5	412.0
3	0.0	64.0	320.0	238.3	374.0	455.0

Table A.12: Insertions into 3 medium sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.0	247.0	609.0	503.9	746.2	1148.0
2	0.0	71.0	589.0	494.3	716.0	1342.0
3 6 0.0	215.5	593.0	517.4	789.5	1256.0	
4 6 0.0	263.5	619.5	517.5	745.5	1197.0	
5	0.0	69.0	574.0	488.5	749.0	1133.0

Table A.13: Insertions into 5 medium sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.00	71.75	325.50	258.70	379.00	517.00
2	0.0	89.0	345.0	272.3	394.0	753.0
3	0.0	138.5	346.0	277.9	406.2	592.0
4 6 0.0	167.5	335.0	266.5	386.0	546.0	
5	0.0	170.0	333.0	271.3	393.0	658.0
6	0.0	160.8	334.0	271.1	390.0	709.0
7	0.0	148.0	329.5	270.6	379.0	641.0
8	0.0	141.2	329.0	269.0	387.2	592.0
9	0.0	145.8	322.5	263.2	374.2	584.0
10	0.0	152.0	322.0	260.8	370.0	539.0

Table A.14: Insertions into 10 medium sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.0	98.0	276.0	230.6	330.2	470.0
2	0.0	51.0	285.0	227.7	326.0	443.0
3	0.0	138.5	297.5	241.4	349.0	454.0
4	0.0	55.0	278.0	225.9	327.0	475.0
5	0.0	123.8	281.0	227.0	330.2	448.0
6	0.0	124.8	280.0	229.8	328.0	465.0
7	0.0	67.0	273.0	229.8	326.2	498.0
8	0.0	117.2	293.0	235.8	331.0	544.0
9	0.0	142.0	293.0	236.5	330.0	574.0
10	0.00	86.75	264.00	219.50	318.00	459.00
11	0.00	90.25	259.00	215.50	306.00	624.00
12	0.00	97.25	284.50	227.70	326.80	600.00
13	0.00	81.75	290.00	229.50	324.50	455.00
14	0.0	65.0	286.0	233.5	338.0	454.0
15	0.0	66.0	300.5	243.7	348.0	545.0

Table A.15: Insertions into 15medium sharded MongoDB. 5 sec sample frequency.

Large instances

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.0	329.5	849.0	754.4	1009.0	1986.0

Table A.16: Insertions into 1 large regular MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.0	346.5	852.5	710.0	985.0	1460.0

Table A.17: Insertions into 1 large sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.0	152.8	458.5	374.9	527.0	723.0
2	0.0	129.5	449.5	357.0	510.2	732.0

Table A.18: Insertions into 2 large sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.0	385.0	887.0	783.5	1117.0	1867.0
2	0	478	1026	904	1270	2472
3	0.0	470.0	1004.0	874.9	1240.0	1872.0

Table A.19: Insertions into 3 large sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
1	0.00	57.75	215.00	171.40	249.00	296.00
2	0.00	52.25	218.50	170.00	249.00	328.00
3	0.0	55.0	226.5	175.2	256.2	328.0
4	0.0	55.0	226.5	175.2	256.2	328.0
5	0.0	49.0	224.0	172.7	258.0	318.0

Table A.20: Insertions into 5 large sharded MongoDB. 5 sec sample frequency.

Distribution of revisions inserted						
Instance	Min.	1rd Qu.	Median	Mean	3rd Qu.	Max.
0.00	40.00	122.00	98.09	144.00	254.00	
0.00	32.25	138.50	107.50	157.80	230.00	
0.0	32.0	119.0	95.5	141.0	212.0	
0.0	34.0	126.0	104.4	155.0	276.0	
0.0	35.5	128.0	103.5	151.0	265.0	
0.00	32.25	117.00	95.36	143.00	223.00	
0.00	32.25	126.00	98.37	144.80	202.00	
0.0	35.0	134.0	106.1	156.0	240.0	
0.00	32.00	121.00	97.68	140.00	270.00	
0.0	32.0	132.0	104.7	156.0	222.0	

Table A.21: Insertions into 10 large sharded MongoDB. 5 sec sample frequency.