DEGREE PROJECT, IN DEGREE PROGRAMME IN COMPUTER ENGINEERING
FIRST LEVEL
STOCKHOLM, SWEDEN 2015

SQL and NoSQL databases

A CASE STUDY IN THE AZURE CLOUD

FABIAN MIIRO & MIKAEL NÄÄS

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

SQL and NoSQL databases

A case study in the Azure cloud

Fabian Miiro & Mikael Nääs

2015-06-29

Degree Project in Computer Engineering
II122X

Examiner
Anders Sjögren

Supervisor
Fadil Galjic

KTH Royal Institute of Technology
School of Information and Communication Technology (ICT)
SE-164 40 Kista/Stockholm, Sverige

Sammanfattning

I denna rapport jämförs Azure SQL Database med NoSQL lösningen DocumentDB, under förutsättningarna givna av ett turbaserat flerspelarspel. NoSQL är en relativt ny typ av databas, som till skillnad från dess relationsbaserade kusin har nya lovord angående fantastisk prestanda och oändlig skalbarhet. Det sägs att det är skapt för dagens problem utan gårdagens begränsningar.

Warmkitten vill ta reda på om löftena kring NoSQL håller i deras scenario. Eller om den nuvarande SQL lösningen är en bättre match för deras spel med tanke på skalning, snabbhet och anvädningen utav .NET.

Jämförelsen gjordes genom att först skapa ett testramverk för spelets flerspelar funktioner. Sedan optimerades både den relations- och den icke-relationsbaserade lösningen, baserat på rekommendationer och experters utlåtande. Testerna kördes sedan under förhållanden som skulle återspegla verkligheten.

Rapporten visar att denna nya typ utav databaser helt klart är mogna nog att i vissa scenarion användas istället för de relationsbaserade databaserna, givet att tillräckligt mycket tid ges åt att förstå skillnaderna mellan databastyperna och vad det innebär. Extra betänketid bör läggas på hur NoSQL skalar samt vilka skillnader som det innebär när relations-begränsningarna försvinner. Prestandamässigt så är DocumentDB snabbare under normal användning när det finns lugnare stunder, SQL är bättre för applikationer som behöver köra under långa stunder med hög användning utav databasen.

Rapportens slutsats blir en rekommendation till Warmkitten att fortsätta med den redan utvecklade SQL lösningen. Detta då prestandaförbättringarna från NoSQL inte väger upp för den ökade transaktionskomplexiteten, då spelet använder transaktioner i hög utsträckning.

Nyckelord

NoSQL, Databaser, DocumentDB, Molnet, Migrering, Tester, Jämförelse

Abstract

This paper was created to compare Azure SQL Database to the Azure NoSQL solution DocumentDB. This case study is done in the scope of a turn-based multiplayer game created by Warmkitten. NoSQL is a new type of databases, instead of the relational kind we are used to, this new type gives promises of increased performance and unlimited scalability. It is marketed that it is created for the problems of today and not bound by yesterday's limitations.

Warmkitten would like to gain insight if the promises from NoSQL holds for the game they are developing or if the current SQL solution fit their needs. They are most interested in the areas of scaling, performance and .NET interoperability.

The comparison was carried out by creating a test suite testing the game functionalities. Then both the SQL and NoSQL solution was optimized based on best practices and expert guidelines. The tests were then run under circumstances mimicking real-world scenarios.

The paper shows that NoSQL is a valid replacement to SQL, if enough time and thought is put into the implementation. Given that NoSQL is a good fit for the problem at hand. Our paper shows that performance wise NoSQL is faster under normal load when there are times with less load, SQL is better for applications that will have continuously heavy load.

The final verdict of this paper is a recommendation for Warmkitten to continue using SQL for their game. This is because the recorded performance improvements does not outweighs the transactional problems seen in this case study created by the transactional nature of the game.

Keywords

NoSQL, Databases, DocumentDB, The Cloud, Migration, Tests, Comparison

Foreword

This report is a labor of love. We are two Swedish students that have tried our best at sharing our findings in English. Bear in mind that English is not the native tongue of the authors.

Stockholm, June 15
Fabian Miiro & Mikael Nääs

Recognitions

We would here like to recognize some people that have been of outmost importance for finalizing our thesis.

Dag König – Microsoft – Gave a speech about DocumentDB and answered our questions during a community event.

Ulrika Malmgren – MagineTV – Software tester and speaker that emphasized not only the hard subjects but also the soft ones.

Ryan CrawCour – Microsoft – Team member of the DocumentDB team. Author of several articles as well as the host of a full day of DocumentDB training.

Pontus Wittenmark – Warmkitten – Owner and developer at Warmkitten. Provided us with knowledge of the game and helped with programming challenges. A++ .NET Developer.

Henrik Miiro – Swedavia – Working with SQL databases for a substantial amount of time Henrik has provided the most substantial contributions to the SQL knowledge needed for the project. The SQL Query found in the appendix is provided by Henrik.

Table of Contents

List of Figures

List of tables

Abbreviations

PaaS – Platform as a Service

SaaS – Software as a Service

SQL– Structured Query Language

CRUD – Create Read Update Delete

NoSQL – Not Only SQL databases

JSON –JavaScript Object Notation

SQL – Structured Query Language

T-SQL – Transact SQL

RU – Request Unit

DTU – Database Throughput Unit

ACID – Atomicity, Consistency, Isolation, Durability

URI – Uniform Resource Identifier

1 Introduction

There has always been a need to store and process information (data) when dealing with digital applications. We have come a long way since the first hard drives and today there are a lot more alternatives to just storing data on disk. Previously databases of the relational kind has been seen as the de-facto standard. Now a new breed of databases under the umbrella term NoSQL is emerging, these new databases are built for performance and scalability at its core.

Our task was to compare the two database solutions when using SQL Server as the relational solution and DocumentDB as the NoSQL solution.

1.1 Introduction to databases

Traditionally when tasked with the problem of storing data for analysis, access and querying, the solution was to use some sort of structured data model within your databases called relational databases. These databases have a very specific structure and needs to be clearly specified before use. The kind of data needs to be specified prior to usage, which will lock the database with constraints.

A recently emerging trend in IT, specifically in Big Data [1] is to use one of the many new unstructured database models that currently reside under the term NoSQL. At one point NoSQL stood for "Not Only SQL" [2], the definition by Fowler however includes the limitation that NoSQL databases should not use SQL (Structured Query Language). Because DocumentDB accepts SQL queries this definition did not fit us, instead we are using the abstract definition by Margaret Rouse: "NoSQL, which encompasses a wide range of technologies and architectures, seeks to solve the scalability and big data performance issues that relational databases weren't designed to address." [3].

There are four different types of NoSQL databases. These are Key-value, Document, Column and Graph based [4]. These will be summarized below.

1.1.1 Key-Value Store

A key-value store is the simplest way of storing and retrieving data. It is constructed of one key and then a value together. The key is a simple string value used to get the value of that key. The keys are the only thing that is being indexed. The value can be any arbitrary data, whether that might be a single value, a hierarchical structure or a list. The value does not need to be structured and pre-defined as in a relational database [5].

1.1.2 Document-based Store and DocumentDB

A document database stores multiple documents in collections for later use. Much like the key-value store, each document has a key to query by to get the right information. The difference with a document-based store compared to the key-value store is that there is a possibility to index everything inside the documents. This solution is still similar to the key-value store in the sense that arbitrary data can be stored in each document. There might be some rules that govern if the

data can be indexed, but there are no constraints on how the data should look and the hierarchy of the data [6].

The database we are going to use was chosen by Warmkitten and is called DocumentDB, it is a document-based store developed by Microsoft that is being sold as a service through their Azure cloud solution. It has been developed by Microsoft to, according to them, solve a couple of problems they found they had with the current document-based stores. This includes that DocumentDB has a high focus on tunable consistency, it is also completely schema-free [7].

One of the bigger differences between DocumentDB and the other available solutions is that DocumentDB accepts SQL queries. This might not seem like a big deal, but if we do a comparison to the most popular (based on search data) [8] document based database MongoDB we can see that there is a big difference between the syntax used. Using a translation tool we translate a SQL query consisting of four rows to its MongoDB equivalence, as can be seen in Figure 1 there is a big difference [9]. This automatic translation is done by a company specialized working with data storage. MongoDB is one of their data areas, there is no obvious reason for them to let MongoDB look inefficient so we trust that the translation is as favorable as can be, given the companies knowledge.

SQL Query:

```
SELECT person, SUM(score), AVG(score), MIN(score), MAX(score), COUNT(*)
FROM demo
WHERE score > 0 AND person IN('bob','jake')
GROUP BY person;
```

MongoDB Syntax:

```
db.demo.group({
    "key": {
        "person": true
    },
    "initial": {
        "sumscore": 0,
        "sumforaverageaveragescore": 0,
        "countforaverageaveragescore": 0,
        "countstar": 0
    },
    "reduce": function(obj, prev) {
        prev.sumscore = prev.sumscore + obj.score - 0;
        prev.sumforaverageaveragescore += obj.score;
        prev.countforaverageaveragescore++;
        prev.minimumvaluescore = isNaN(prev.minimumvaluescore) ? obj.score : Math.min(prev.minimumvaluescore
        prev.maximumvaluescore = isNaN(prev.maximumvaluescore) ? obj.score : Math.max(prev.maximumvaluescore
        if (true != null) if (true instanceof Array) prev.countstar += true.length;
        else prev.countstar++;
    },
    "finalize": function(prev) {
        prev.averagescore = prev.sumforaverageaveragescore / prev.countforaverageaveragescore;
        delete prev.sumforaverageaveragescore;
        delete prev.countforaverageaveragescore;
    },
    "cond": {
        "score": {
            "$gt": 0
        },
        "person": {
            "$in": ["bob", "jake"]
        }
    }
});
```

Figure 1 showing a SQL query translated into MongoDB syntax

1.1.3 Column-based Store

A column based store is in a way much like a relational database in the sense that both of them have a clear notion of rows and columns. The big difference is that while a relational model is built to be very dynamic in rows and restricted in columns, a column-based store is dynamic in case of the columns. A column-based store could be used in a similar fashion as the relational database. The difference is that a column-based store can accept input using new columns not already defined. Another differentiator between the two is their use of relations. The relational database creates several tables and then stores relations between the two. A column-based store can store columns inside columns, storing the data closer to the data it relates to [10].

1.1.4 Graph-based

A graph based database is a database that instead of just storing data that is then indexed to one degree or another, it stores data and the relationship between it. The data in a graph database does not need to be as structured as in a relational database but can have some structural roles on how it should look [11]. One very common graph database people are fairly familiar with is the Facebook graph [12].

1.2 Why consider NoSQL

When relational databases were first mentioned in the 1970s [13] a single hard drive could cost as much as 25970$ for 100MB [14]. This is about 7 800 000 times more expensive than data is now (prices adjusted based on inflation) [14] [15] [16]. During that time wanting a data-store system that consumed as little data as possible was of high concern. However, given the extremely cheap price of storage today, relational databases could make way for less storage frugal alternatives. This is still not the case with relational databases still the majority [8] [17]. Here is where the promise of NoSQL lies. Most NoSQL implementations share the notion that they are built for the present, for today's systems, designed to solve the problems of today [18].

Except the promise of being built for the IT world as it is today, NoSQL systems also gives new promises in scalability and performance. The biggest thing here is to scale an ordinary SQL database you often require to scale vertically (adding more power to a single node) whereas NoSQL scale horizontally [19]. This means that a NoSQL implementation is specifically built to be split up into shards and scaled by adding more nodes. This comes with several advantages. One is that adding a new node is often cheaper and more easily to do than upgrading an already running machine, the point of failure is no longer a single place while using horizontal scaling. Another advantage in our new dynamic world is that horizontal scaling means that an application can be easily scaled down without any downtime, you simply remove a node and let the remaining handle the requests, scaling down vertically would in many times require turning off that one machine and replacing the RAM, an action that requires downtime.

Another advantage of NoSQL is its use of schema-free data [7], where indexes are created in real-time and no tables or schemas needs to be created beforehand. Any data can be sent in and can be

queried by the database. This is a very compelling argument if the data used can change over time. A relational structure would require the creation of separate tables for each type of data stored while some NoSQL implementations can handle this directly. This also means that NoSQL can handle nested data, while relational databases need a new table and references to that instead of nesting data. This creates a situation where relational databases needs to query the database multiple times, when NoSQL can get the same data with only one request to a single point.

The downside to the NoSQL solution is that because it is so new and different, a lot of developers see it as an ugly solution because it fundamentally breaks the rules and doesn't use the normalization techniques used by relational databases [20]. One example is that in NoSQL storing multiple copies of data is okay, it is actually encouraged to improve performance and traceability [21]. This is something that in relational databases are seen as wasteful [20].

Several major companies that require a new degree of performance and scalability compared to relational databases have changed to using NoSQL. These include eBay, LinkedIn and Adobe [22] to name a few.

1.3 Warmkittten

Figure 2 Warmkitten logo

Warmkitten is a Stockholm-based company offering consulting services and technical training on the Microsoft stack [23]. Warmkitten has a history of mostly doing consulting work for other companies but with the game "Spellclash" the company is producing something of their own. This game is a major change in how the company usually produces revenue and because of this, there is a lot of discovery and testing that needs to be carried out.

1.4 Problem

The problem could be explained as an uncertainty of whether or not an SQL solution really is the most suitable answer for Warmkitten's game Spellclash and its backend. It is also uncertain whether or not the current relational database is constructed in an efficient way following known best practices.

Matchmaking and user creation are the two big problem areas and Warmkitten is unsure whether or not its algorithm is suited for a SQL or NoSQL solution when taking into account price, performance, scalability and .NET interoperability.

Warmkitten as a company is focused on .NET technologies and it is therefore vital for them that the solution they use should be .NET interoperable.

This project is run on a small budget and keeping costs down are detrimental to the economical solvency of Warmkitten.

1.5 Research Questions

We want to gain certainty in whether or not NoSQL is a better fit for Warmkitten compared to SQL when taking price, performance, scalability and .NET interoperability into account. So our research can be defined by the following questions:

- *While comparing the solutions, which of the two gives faster average request time*
- *Does the SQL solution offer higher .NET interoperability than the NoSQL solution*
- *Which solution requires the least effort to scale*
- *Which solution has the cheapest starting point*

1.6 Purpose

The purpose of this paper is to gain knowledge whether a NoSQL solution using a document-based store is better for Warmkitten compared to a SQL Server solution. Taking into account price, performance, scalability and .NET interoperability.

This will benefit Warmkitten in such a way that they will save substantial development times and ensure maintainability. Warmkitten will also have gained extensive knowledge about NoSQL as an emerging technology and can therefore make an active choice between the two solutions in confidence.

1.7 Goals

Our primary goal is to produce a performance test suit that covers the critical use cases user creation and matchmaking. This test suite will be used to compare the two implementations based on the research questions.

Another goal is to produce the actual NoSQL implementation and finally to create a report.

1.8 Method

The written report was written like a case study using something often called: Simple blogging. Followed by a process we call cleansing.

This two-step process is chosen because it puts a high focus on actual writing at first when we do not need to focus on headings, wording and structure as much. In the early parts of the project, the important thing is to write everything down so it will be remembered and can be worked on later.

During the cleansing process we take what is already written, rewrite parts of it into a more easily understandable manner and makes sure the text is structured, easier to follow and adheres to the examiners standards.

The report uses the project as a case study to provide insight to the general case of comparing SQL to NoSQL. When working with this in mind we have to design our goals to provide valuable information not only to Warmkitten, but also to the community in a larger sense.

1.9 Scope

The scope chosen for this project is limited so as to try and get a clear picture of and to limit the possible variables so that the time given will be enough to produce valuable results.

1.9.1 Inside the scope

When comparing the SQL and NoSQL solution we have decided to measure them using four different metrics. Those are price, performance, scalability and .NET interoperability.

Both implementations will be compared on each metric and a conclusion formed on each. This means that we might not find out which implementation is “the best”. But instead we will find out which solution is best if you look at price, performance, scalability and .NET interoperability. However, a recommendation will be made to Warmkitten based on the individual conclusions made.

We define these different metrics as such:

1.9.1.1 Price & Scalability

The price and scalability of the implementation is going to be compared using some different measurements. First the minimum and maximum prices for each service tier will be compared. Then the prices will be compared based on the scalability they offer. How easy is it to scale and finally how good the prices scale.

1.9.1.2 Performance

Performance is measured by comparing the average time it takes to perform a full test during a thirty-minute period under normal circumstances.

1.9.1.3 .NET Interoperability

.NET interoperability is measured by counting how much “non .NET” code is written for each solution.

1.9.2 Outside the scope

The only thing inside the scope is what is defined above. This means that outside the scope are such things as: security, other NoSQL solutions except for the document-based store DocumentDB and other relational solutions except Microsoft SQL Server. The paper also doesn’t consider backups, data relating to this is therefore not considered in our comparisons.

This scope is also focusing on the game Spellclash which Warmkitten is developing. This means that the report will focused on the scenarios in that game and will not focus on general scenarios. It is therefore outside the scope to give a general conclusion whether or not to use NoSQL in the IT industry. Our conclusion will focus on the multiplayer turn-based game scenario provided by Warmkitten.

2 Theoretical Background

This chapter goes through the theoretical knowledge needed to understand the rest of the paper, such as the individual problems for Warmkitten and how the different phases of matchmaking works.

2.1 Information specific to Warmkitten

This section will explain some SQL problems that Warmkitten are experiencing and why a NoSQL solution might solve this in their game Spellclash. The initial solution was developed in SQL mainly because of familiarity and speed to production.

2.1.1 SQL Problems and NoSQL promises

The identified problem with the current SQL solution is that it needs to store and manipulate data twice in every operation that is connected to a game. An example of this double stored data in need of manipulation is shown in Figure 3.

Because the game natively uses JSON (JavaScript Object Notation) as the game state, which is then saved on the database, some data is stored twice to make sure it can be properly indexed and used in T-SQL (Transact SQL) commands.

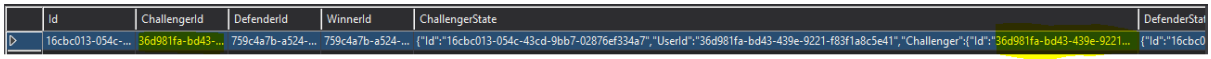

Figure 3 screenshot from the SQL database with double stored data marked

NoSQL instead handles JSON natively and could therefore seem to fit better. In such a solution logic wouldn't be required to go through the data twice, update it once to make it correct, and another time to make the new information included in the index.

2.2 The game

Figure 4 showing the spellclash game with a player "WopWop" playing against the AI opponent

The game is a turn-based game with multiplayer components. It can be compared to chess in the same regard that you have a square playing field as seen in Figure 4. Different from chess is however that every turn the player isn't constrained to moving only one piece but can move all pieces every turn. Instead the limitation is that every character has a predefined amount of steps it can do each turn (no diagonal steps are allowed). Another differentiator is the games use of spells that are either one time spells or use up allotted resources (mana). These spells can in different ways affect the playing field like creating a boulder or spawning a new character.

The multiplayer parts of the game use several components to create the full experience as imagined by Warmkitten. This includes the ability for players to have a profile of their own, where they can gain jewels by playing. A matchmaking system is also created as well as a system to block other players, effectively eliminating the possibility to get matched with that player.

2.2.1 Player

Figure 5 A PlayerTag and the jewels that account holds

In Spellclash a player profile holds several important pieces of information to make the game work. The profile contains some public information. Such as the name of that player (can be made up), and the amount of jewels that player has amassed. This info is visible in the game for both players and looks like Figure 5. Privately the user profile also holds important information such as the user Id and the users token. This token is what is used to validate the user to the multiplayer service so that no one else can perform actions in another player's name.

2.2.2 Game instance

A multiplayer game on the server holds several pieces of information that are all required to play a game between two players. This includes the Id's of the challenger and defender as well as the current state of the game, the stage the game is played on and which types of rules the current match uses.

2.2.3 Matching process

To match two users together and enable them to play against each other there are several steps that needs to take place, this sub-section will outline those steps.

2.2.3.1 Game creation

If a player chooses to play against a random opponent and there is no game available that matches the players criteria's, the client will be instructed to start a new game where the player will be the

challenger. This player will not yet know the identity of the defender which they should play against. Not until after the player has submitted the initial turn a new game is created on the server that another player can match against.

Every player can also freely create new games, where they can specify their own game parameters.

2.2.3.2 Game find

If a player chose to play against another random player and there is already a game available without a defender, the player takes that defender position and will see the first turn as it was made by the player that created the game. This game is now matched. When this newly assigned defender submits their turn the game is considered as ongoing.

2.2.3.3 Matching problem

The problem during the matching process is a problem of concurrency. However, this problem cannot be solved by a simple mutex, since this application has to be able to scale and run on multiple nodes simultaneously.

The problem arises if there is a game available to match against and two players enters the matching process. This pseudo code (also illustrated in Figure 6) then handles the request:

1. Validate user
2. Find games available for matching
3. Put user as the defender
4. Return a game for the user to play

This might seem quick and simple. The problem is that when more than one user does this at the same split second, both can find the same matching game before anyone of them have the chance to take the spot as defender.

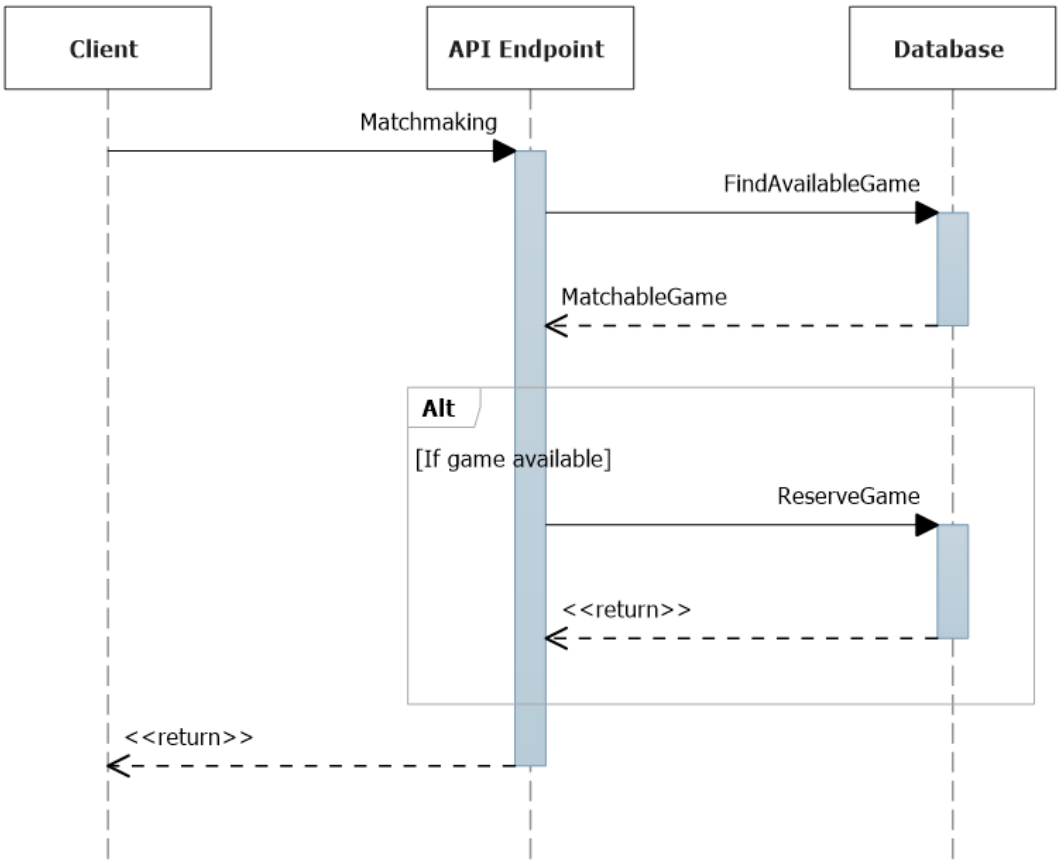

Figure 6 a sequence diagram of the matchmaking process

2.3 Cloud Billing

When talking about and comparing different solutions that are being consumed as a PaaS (Platform as a Service) the billing module is very different from traditional licensed software and bought hardware. Instead of buying licenses for a specific number of users or for a certain hardware you only pay for what you use. This is sometimes called “Pay-as-you-go” [24].

2.3.1 SQL Server

When buying and using the Azure SQL Server service, what you end up doing is reserving hardware resources that you think are needed for your application. The billing is done hourly and each database can be scaled through three different tiers: Basic, Standard and Premium [25].

The variables that change during the different tiers is throughput, size and point in time restore.

Throughput is measured using Database Throughput Units (DTUs). DTUs are a unit provided by Microsoft that is a relative unit showing the performance of each database. It is affected by CPU, RAM and I/O speeds of the underlying storage layer. The DTU unit is used to indicate the performance of the hardware as a black box since we can’t see the actual specifications [25].

Table 1 shows data about each SQL offering in Azure [25].

Table 1 Prices for SQL server tiers as recorded on 2015-05-16

Tier	DTU	Size (GB)	Price / Hour (SEK)
Basic 0	5	2	0.045
Standard 0	10	250	0.1356
Standard 1	20	250	0.2706
Standard 2	50	250	0.6767
Standard 3	100	250	1.3533
Premium 1	125	500	4.1954
Premium 2	250	500	8.3907
Premium 3	1000	500	33.57

2.3.2 DocumentDB

The basic idea behind DocumentDB billing is that, much like the SQL Server variant, performance is reserved for you. What you reserve is RUs (Request Units) and size.

Request Units is a measure for the resources required to perform database operations. But instead of reserving this hardware directly as in the SQL Server example the RUs are measured per second basis. This means that the actual speed of which operations are performed will always perform at maximum speed and will never decline, as they might when Azure SQL becomes heavy loaded. Instead calls done against the service when the RU quota has been spent are not performed [26]. This creates predictable performance and the scaling is simply an increase in operations per second.

Instead of databases you buy “Collections”, these collections exist within a database and add RU / second and size to that database. It is worth noting that query execution and transactions are bound to the collections it is performed on [27].

Table 2 Prices for DocumentDB collections as of 2015-05-16

Tier	RU / second	Size (GB)	Price / hour (SEK)
S1	250	10	0.229
S2	1000	10	0.45

S3	2500	10	0.90

2.4 Cloud Scaling

In this section it will be discussed how the two different solutions scale.

2.4.1 SQL Scaling

SQL Databases can be scaled vertically by switching the tier or performance level it is in. This means that a SQL database can scale in tiers from 2GB in size and 5DTUs up to 500GB and 1000DTUs. However, changing the service tier of a database may require the database to be copied internally. If this happens it may take from a few minutes to several hours depending on the size of the database [25].

There's also restrictions on how often performance level and tier can be changed, which is currently 4 times per 24 hours [25].

Up until the 2015 BUILD conference a SQL database was a single isolated instance. However at BUILD Microsoft introduced the elastic database tools that allows data to be added and queried over several databases, effectively adding automatic scale-out functionality [28].

2.4.2 NoSQL Scaling

The NoSQL solution DocumentDB has three tiers, all sporting 10GB of storage each. The difference between the SQL solution is that you do not scale and add databases per-se. Instead collections are scaled and/or added to a database. A collection is very much like a database, every collection has its own RUs and size and is also isolated in the sense that stored procedures are locked to one collection and can't affect another collection [29] [26].

Compared to SQL the NoSQL service is built for sharding and changing of tiers. There is no limit on how often tiers can be changed and there's no noticeable delay when changing of tiers since it is only a measurement of reserved throughput [29] [30].

2.5 Visual Studio Test tools

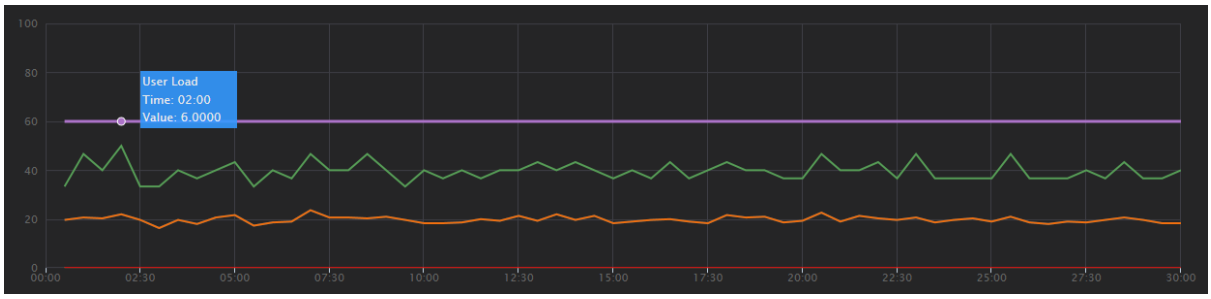

Figure 7 A load test running with 6 virtual users, showing response times and pages per second

To test and gather data about implementations, tests needs to be created and run. Instead of handling all the parameters connected to this ourselves we use tools built into Visual Studio.

2.5.1 Web Performance Test

Each web performance test is a repeatable test that does one or more web requests, potentially validating the response in some way. This is shown in Figure 8 where “ResponseCode” is validated and used as a conditional element to decide the next step in the test run.

A web performance test come in two shapes, these are “Web Test” or “Coded Web Test”. The difference is that the “Coded Web Test” is managed using code, while the “Web Test” is created and managed using a visual representation. The Coded Web Test is normal .NET code and is therefore not as limited as the UI created web tests. Coded Web tests can be generated from non-Coded Web Tests from within the test editor [31].

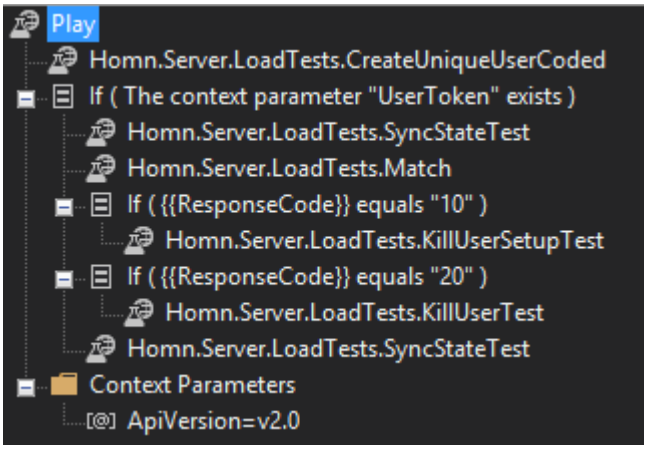

Figure 8 a web test using several conditional Coded Web Tests

2.5.2 Load Test

A load test is a collection of one or more Web Performance Tests. A load test is designed to test the performance of a site or application by running one or more tests multiple times with multiple virtual users in parallell, to gather data that can be used for improvements in the application [32]. An example of a running load test can be seen in Figure 7.

2.6 Related works

This section explains some of the related works found during our searches. It also outlines some simple ideas we found and what we are taking with us from them.

2.6.1 Introducing DocumentDB - A NoSQL Database for Microsoft Azure

This is a whitepaper regarding DocumentDB. It is a clear and focused introduction on what it is and what it does. It explains some of the decisions that have been made in forming this document database and will therefore give us clear explanations regarding the DocumentDB implementation. We like this, it gives us the opportunity to not dig too deep, but to have already researched and backed up arguments provided to us [33].

2.6.2 SQL databases v. NoSQL databases

This article deals with whether or not SQL actually is slow or if it is the effects of keeping SQL ACID (Atomicity, Consistency, Isolation, Durability) that is keeping speed down. The paper compares and contrasts SQL and NoSQL and discusses speed. [34] [35]

2.6.3 MySQL to NoSQL: data modeling challenges in supporting scalability

This article explains the differences when modeling for SQL and NoSQL and brings up problems and solutions with migrating an existing solution in SQL to a NoSQL Solution. [34]

2.6.4 A performance comparison of SQL and NoSQL databases

Manoharan & Li talks about performance differences between the different NoSQL Databases and SQL, it does not talk about transactions and the performance that deals with ACID. The paper shows that we do not have to compare the different NoSQL Solutions since they are all very similar in the results they produce. [36]

3 Methodology

During the project we have three distinct phases in the case study as well as a final optimization phase. These are defined to limit ourselves in our scope and make sure we do not stick to one of the phases for too long.

3.1 Phases

This section will describe the different methodologies used in this case study.

3.1.1 Test creation

The problem is approached by first developing a comprehensive test suite for the solution that is currently in place. This is to be able to create measurements that can be used to easily see whether the suggested NoSQL solution is better suited for this application or not by simply running the test suite and comparing the results. The test suite will more specifically help to discern if the NoSQL solution is faster than Warmkittens current solution of having a relational database using traditional SQL.

This will also bring with it a great deal of positive consequences that can be used further down the line for Warmkitten and the project when developing the NoSQL solution, and also to improve on the existing solution. Optimizing the existing solution is done to ensure that the increased efficiency is because of our migration to NoSQL, and not because the previous implementation was inefficient.

The testing suite is to be made with high quality in mind since this will provide the bulk of our argument for or against the switch to a NoSQL solution.

3.1.2 Optimizing existing SQL

If the tests constructed in the previous phase should hold any value when developing the NoSQL solution, the initial SQL solution must be optimized so that the tests will provide a benchmark for the performance of the NoSQL solution and its validity as an improvement for the current solution. To make sure that we are not testing against a inefficient solution we talk to industry experts regarding SQL implementations and follow their advice. The optimization task is constrained by time and optimization tasks are being prioritized based on estimated impact gained from the experts talked to and our own judgment.

We cannot change the underlying infrastructure of the service and will therefore not focus on those things. Instead we focus on things that are in our control. This means that we do not focus on optimizing the database engine or hardware, but our interactions with the database and its configuration.

3.1.3 Implementing a NoSQL Solution

As NoSQL is a reasonably new technology to the world and us, a bigger emphasis is put on research and preparation. This phase is influenced by what was achieved in the previous phases, as they set the benchmark of what should be achieved by our NoSQL solution for it to be better.

Preparation and research was done as a prelude to the iterative process (described in 3.2). Microsoft held a presentation of how and why to develop with DocumentDB that provides good information on the solution and makes some promises on the effectivity of DocumentDB [37].

3.1.4 Further optimization

After the initial phases of tests and implementation we collected the first set of data for our empirical studies. This was in one sense a success since the solution worked, had all elements needed and was accepted by our tests.

The problem however when comparing two solutions to each other is that it is very hard to know if the solution is optimal. There are always things to try, adjustments to be made and sub-seconds to save. The hard part is knowing when to stop.

Our philosophy with this problem was to first implement the minimum set of functionality to make the solution work, to have a solid skeleton that did what it should. But didn't need to be focused on optimization. So after we had gone through all phases one time, we then focused a bit more on optimization. We did this by asking ourselves questions such as "do we really need this request?", "is this necessary?" and "Could we model it another way to make it faster?".

3.2 Phase stages

During each phase we go through a couple of different stages which are going to be described in the following section, with emphasis on the information gathering aspects of it.

A visual representation of this iterative process can be found in Figure 9.

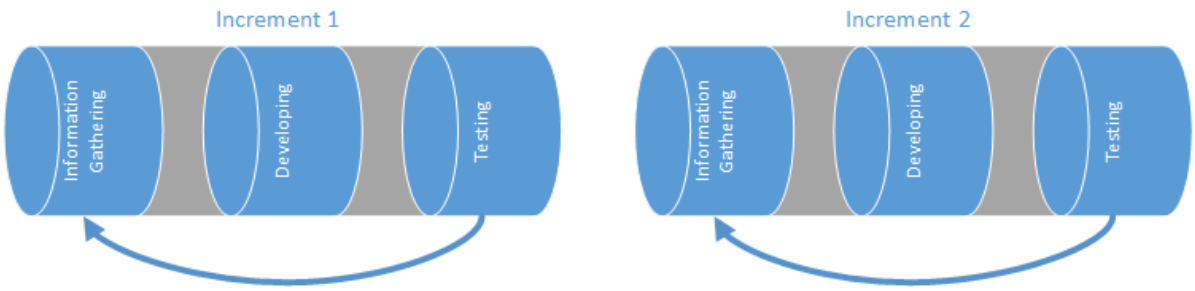

Figure 9 visualizing two increments in our incremental development model

3.2.1 Information Gathering

We are using two methods for information gathering. These are chosen to get trustworthy answers from the industry experts as well as being able to ask questions directly.

3.2.1.1 Literature Searches

To learn more about subjects and technologies we gather information by searching through available literature. This literature could be things such as: Documentation, Blog posts, Forums, News articles.

We have also decided to include podcasts in this category, because of its one-part communication structure and the fact that it is (in the same way as the others) already pre-published. There is no real difference about the information sent or how we can interact with it except the obvious fact that one is in audio while the other is in text format.

3.2.1.2 Talking with people

To get extensive answers to our specific questions we've found local experts and asked them about more specific issues, such that aren't answered in general articles or found in literature searches. We have also used this method to verify that our approach is not going against any industrial best practices unless we are aware of it.

3.2.2 Developing

After gaining enough knowledge to be able to work through the current phase, we start writing the actual implementation. This could be testing code, optimization code or simply implementation code.

We use an incremental model for development where the Architectural Design is included in the iterations. This plays well with the test driven goals of the project, as performance tests can be run iteratively and provide a benchmark of how good our current (the current iteration) solution is. The process was chosen in collaboration with senior developers at Warmkitten.

3.2.3 Testing

The final stage of each phase is to run the full test suite again to be able to see the impact of the phase. The testing stage is also the final accepting stage of every phase. A failing test at this point starts the phase over to make sure that we do not proceed without a working solution.

3.3 Empiricism

This section focuses on the data gathered from our tests. We describe the collected data and why those specific data points were recorded.

3.3.1 Why we record

Among the different things we compare between the two implementations our tests are best suited to get data that we can use to test the performance of the two implementations.

This is particularly good at being proven using empirical studies and we therefore decided that we needed to record and export data from our tests to be able to test the two implementations from a performance standpoint.

3.3.2 What we record

These tests were focused on showing the performance of the solutions, so the tests were set to run a specific time and the average time for a test to run was calculated using the time run and the total tests performed. This data was then saved and is what is used to analyze the performance of the two implementations. The data that we are concerned with during testing is the response time for the user, we use this value to measure performance. To gain understanding of the result from a scalability standpoint we ran the test for a prolonged time so that there was enough data created to make a difference. This time was calculated using the un-optimized SQL solution to find out where we could see a clear performance decrease pattern.

3.3.3 How we record

Using our tests and the two developed solutions we collected data to use as empirical data. The tests were run from the same machine, using the exact same settings except for if the SQL or NoSQL solution should be targeted.

The data was collected by running our tests in Visual Studio and then using the built-in excel exporter to get the raw data from the tests to provide data for the graphs and tables.

3.4 Empirics

All data underneath and in chapter 4 Analysis is the result of the test runs run according to specifications described in section 3.3 Empiricism. The tests that are being run is the test suite described in the testing section. All graphs use thirty seconds sample point except Table 3 that averages the total tests.

Table 3 showing the average times across a thirty-minute test divided by URI

Request Name	**NoSQL**	**SQL**	**Percentage increase**
NewGame	0.163666668s	0.194298253s	11.5306394108082%
SubmitTurn	0.171894422s	0.216944447s	24.5582588009397%
NewOrJoinRandom	0.084048646s	0.130815788s	35.7503807840584%
Sync	0.059793317s	0.12071431s	50.4670848487933%
Create	0.087305904s	0.128711679s	32.1694002546122%

4 Analysis

This chapter works as an umbrella chapter for the three analysis sections. Covering price & scalability, .NET interoperability and performance providing valuable information for the conclusion.

4.1 Analysis | Price & Scalability

This section will analyze gathered data concerning price and scalability. Trying to answer the research questions "*Which solution requires the least effort to scale*" and "*Which solution has is the cheapest starting point*".

4.1.1 Comparing size

Both Azure SQL server and DocumentDB handles size in the same unit of measure "GB" [26] [25]. Therefore, we can quite easily compare the two against each other.

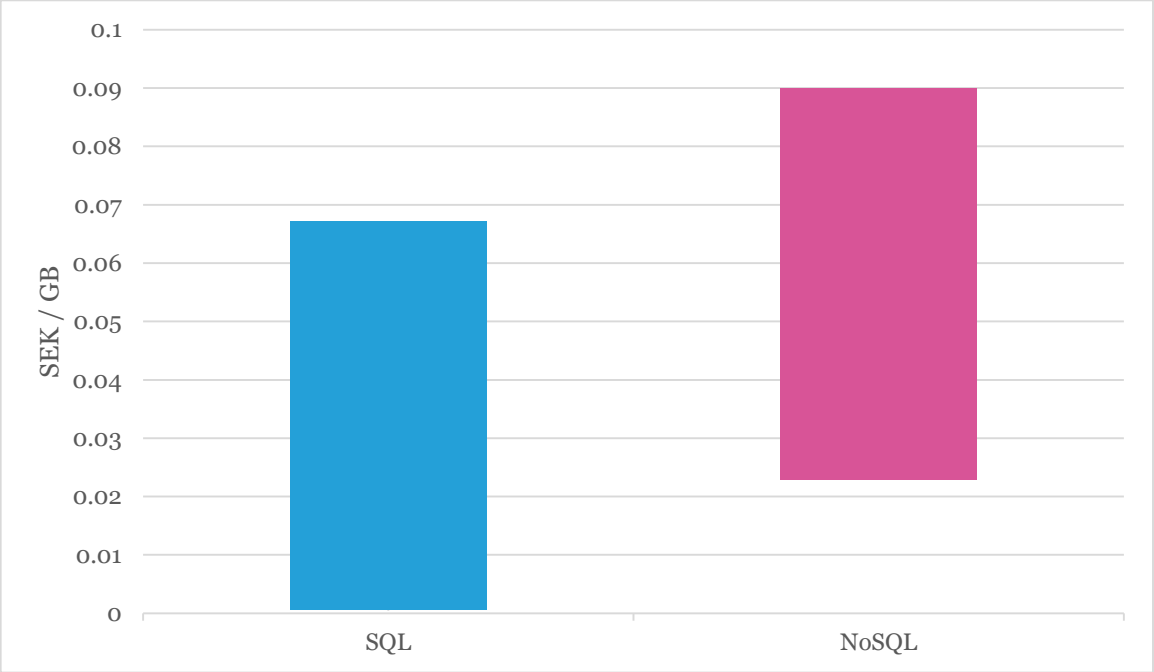

Figure 10 Max and Min price per GB for both NoSQL and SQL

By dividing the price of each selection with the number of GB offered we get the price per GB for each offer. From this list the max and min is extracted which is visualized in Figure 10. We can see that both services mostly overlap. However, the SQL service have the cheapest price per GB and NoSQL has the most expensive when comparing the two.

4.1.2 Comparing price effectiveness

In this sub-section we analyze how effectively the scaling is based on the price. Does double price mean double performance?

4.1.2.1 SQL

With the pricing for the SQL service we can see that the pricing model in relation to DTUs given closely resembles two linear equations (visualized in Figure 11). For the standard tier that can be approximated to: DTU = Price(SEK) · 73.90983. And the premium tier can be written as: DTU = Price(SEK) · 29.78764. Figure 11 is the visualization of drawing both the DTUs for a given price and the price estimates.

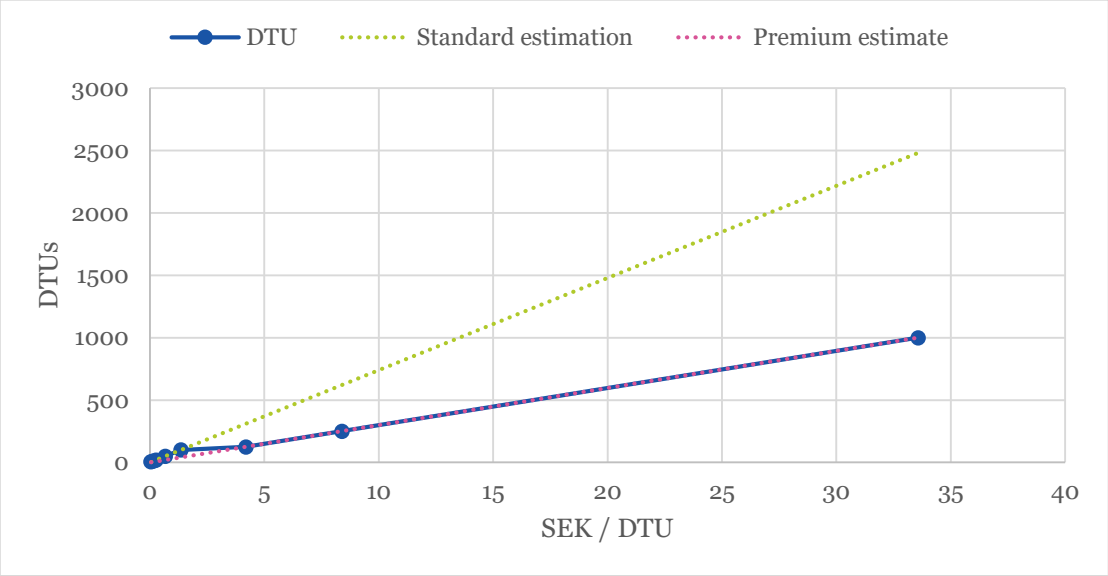

Figure 11 Price of a particular number of DTUs together with the estimation lines

The price for each DTU is placed between 0.009 and 0.033570.

4.1.2.2 NoSQL

The NoSQL pricing model in relation to RUs closely resembles the linear equation

$$RU = Price(SEK) \cdot \ 3353.204 - 517.8838.$$

This is visualized in Figure 12.

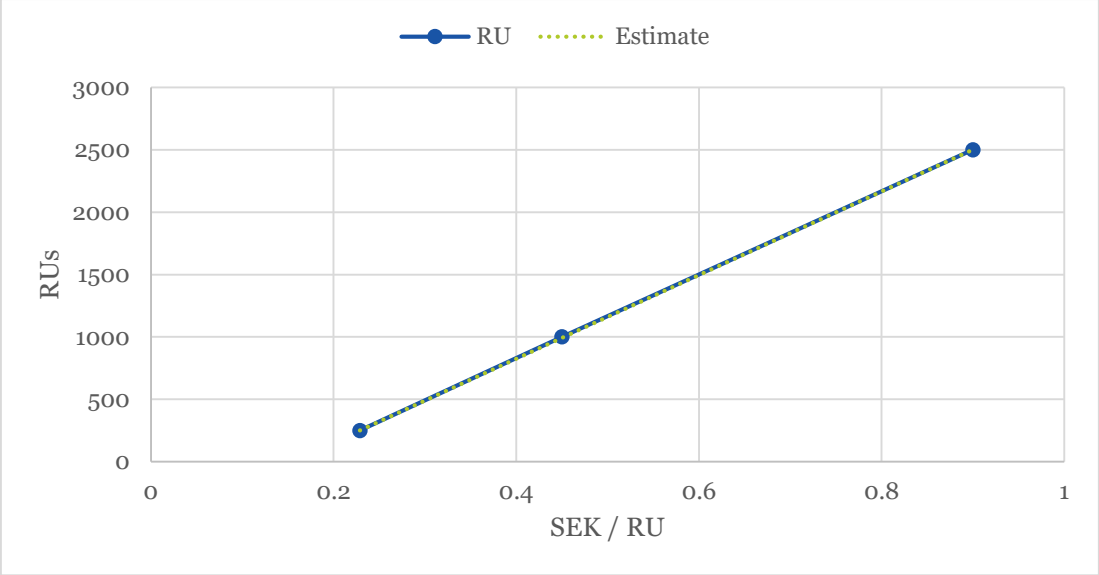

Figure 12 Price of a particular number of RUs together with the estimation line

Each RU is priced between 0.000916 and 0.00036.

4.1.3 Comparing RUs and DTUs

In the theoretical background in section 2.4 cloud scaling is described, both solutions are represented.Neither the difference nor the advantage of each is brought up. This sub-section serves to analyze both approaches and find positive/negative effects of both. If we simply go by price and using the average price of a DTU vs a RU without taking storage into account, then you can buy 25.5996 RUs for each DTU. This is not a surprise since a single RU represents the processing capacity required to read a 1KB document consisting of 10 unique property values where everything is indexed. And at the same time 5 DTUs is the smallest viable database [38].

The two units are otherwise really hard to compare because one measures hardware reserved for usage and the other throughput and processing power that can be used during each second [38]. This means that in the RU case the actual performance is never limited, it is just how much it is used that is limited. In DTUs it is the other way around. You can use it how much you want, but since hardware is the limiting factor it might become slower if it is heavily used.

4.2 Analysis | .NET Interoperability

This section analyses the amount of non .NET code in both our final implementations. Trying to answer the research question "*Does the SQL solution offer higher .NET interoperability than the NoSQL solution*"

4.2.1 What to count and how

We decided to only count the actual files that change between the SQL and NoSQL solutions. This means that the only analyzed classes are those that actually do calls to the database or in it. This include the actual API service class and the stored procedures.

To count the lines of code we are using the regular expression found in appendix 8.2 on the source file. This is used instead of just finding new lines because that would also count the comments, something that we are not interested in.

To count the non .NET interoperable code we simply analyzed the code and counted the lines of code that are executable, but not natively understood by the .NET runtime.

4.2.2 SQL

The SQL solution is in total 898 lines long. If only code rows are counted it adds up to 698. Which means that 200 lines are comments and blank rows.

The only code in the SQL solution we could find that is not .NET interoperable is the one line that includes the raw SQL query.

4.2.3 NoSQL

The NoSQL solution utilizes multiple stored procedures that are written in JavaScript which is a big downside as far as .Net interoperability is concerned. Although convenient when handling JSON objects it doesn't adhere to Warmkittens best practices.

This implementation has a total of 1040 lines where 780 of those are code lines. Which means that 260 of those lines are comments and empty rows.

The code that is not .NET interoperable in the NoSQL implementation are the stored procedures. Those are 82 rows of non .NET interoperable code.

4.2.4 Analysis visualized

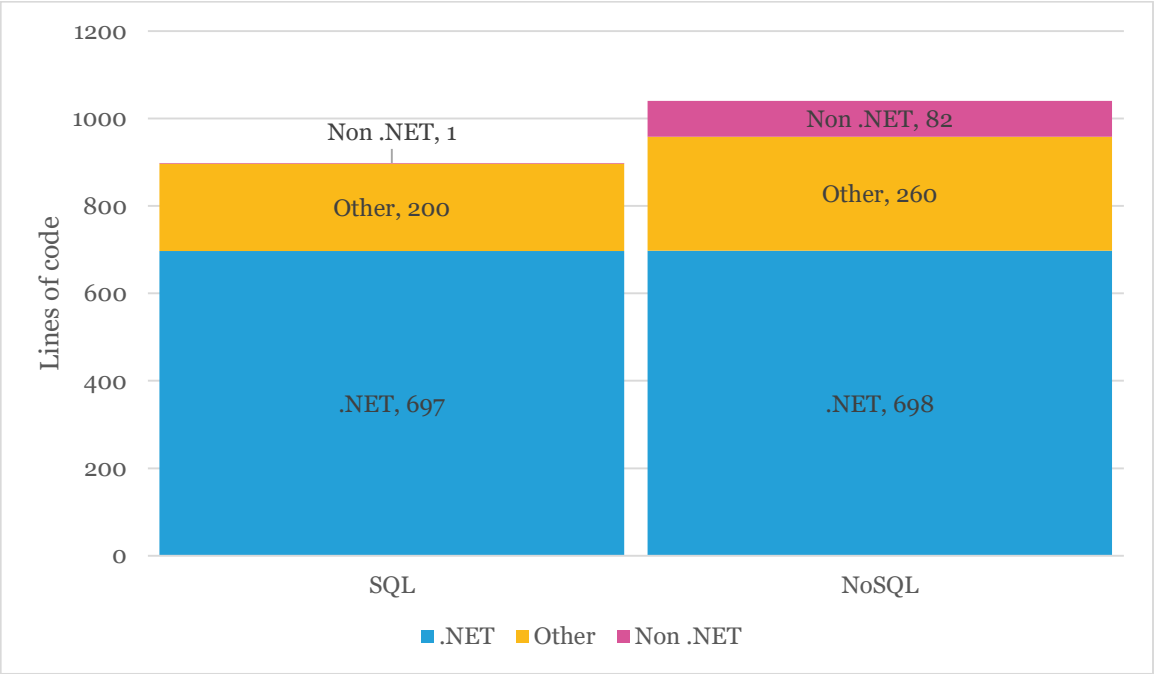

Figure 13 Visualization of the code for both solutions, showing the .NET and non .NET code. NoSQL contains more non .NET code

4.3 Analysis | Performance

This analysis focuses on the research question "*While comparing the solutions, which of the two gives faster average request time*".

4.3.1 Writing tests

Tests are written with Visual Studio and are run locally during development and in the cloud when gathering test data. We do this because of the decrease in transfer time and to remove any variables that could be existing within our environment. Since performance of general usage is what's interesting in this report these are carried out in the form of Load tests with nestled performance tests.

4.3.1.1 Test structure

The tests we run are created to imitate a high load of users on our implementations. Each load test can include several performance tests that are either a Web Test or a Coded Web Test. Illustrated with below list.

- Load tests
    - Performance tests
        - Web Tests
        - Coded Web Tests

4.3.1.2 Creating users

These tests are very important in development not only because it is a big part of the functionality but also because it makes it possible to create a streamlined testing scenario.

This test calls a web service using HTTP POST with a username (PlayerTag). In our tests we provide the username as a GUID to guarantee uniqueness. This test also saves the response code in the test context for later use by the other tests in the suite.

There are two types of creating users tests that we have handled during this project. For performance and throughput it is necessary to create a high volume of users to see where the server handles these requests the slowest. For this type of issue we create random unique users.

4.3.1.3 Playing the game

These tests are composites that are made up of two parts where one part plays the loser and one the winner. This is to simulate the logic of the most frequent use case. The functionalities that are simulated in this test are:

- Matching users against each other
- Creating and joining games
- Submitting turns
- Ending game logic (who lost/won, jewels and time of end)

This is one of the most complex tests.

4.3.1.4 Syncing

The test is rather simple but what it tests is a very complex functionality that both handles roaming between platforms and keeps the state of all of the users games. This is one of the most frequent use cases if not the most frequent use case. The tests are composed of HTTP POST calls to the sync functionality that includes the users tokens, the current games and turn index.

4.3.1.5 Putting it all together

The tests are put together in a web performance test, this makes up our test scenario of creating users that plays games against each other with syncs occurring after each game played. This test plays up one of two scenarios depending on whether or not a game already exists and is waiting for a player.

The test creates a user and a user token is returned to the test. The test then tries to play the game using the find a random opponent functionality in the service and handles the response code. Depending on the response code that tells the test whether or not there is an existing game that is waiting for a second player, the test fills that requested role (being the defender or the player). The test then submits a different turn depending on the role. After both parts of the tests has submitted their turns, each individual test will sync, complete and be marked as a successful test iteration.

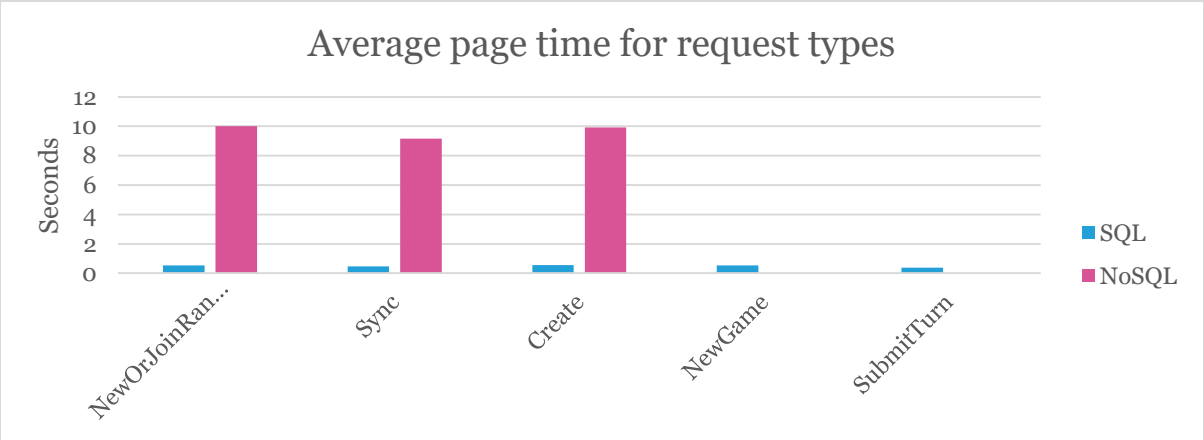

Figure 14 Showing in comparison extremely high values from NoSQL on some and no values from others. SQL consistent low values compared. Data extracted after running unnatural tests

After creating all tests and having a reliable load test that could be run, we realized while looking at the test suite created that the scenario that we now had created was nothing like real life. A service such as this will never have that kind of traffic as we had modeled. There will be pauses and there will be time in-between requests. This lead to the creation of an updated test suite where we included random think times between the tests. That way a more natural scenario had been created, while still recording multiple data-points of interest. None of the previous work was lost. The clear difference between having a test suite that mimics the real world can be seen when comparing Figure 14 and Figure 15.

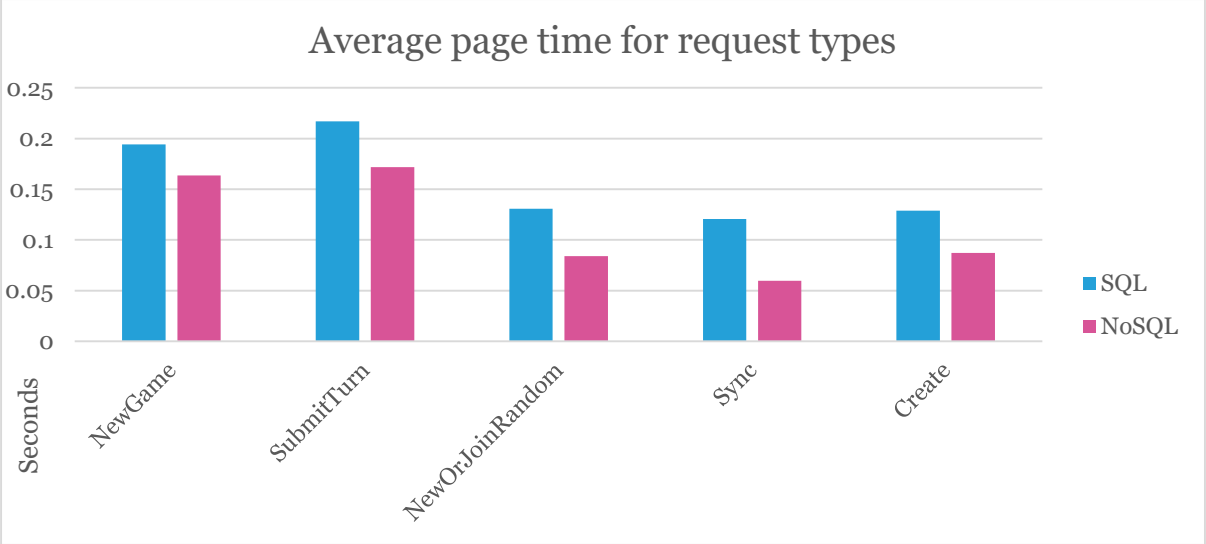

Figure 15 Bars showing the average request times between the SQL and NoSQL solution using the natural scenario, NoSQL takes less time in all request types

4.4 Optimization of SQL

When optimizing a database solution, it is important to identify where your problem area lies, where your request take the longest i.e. where there is most room for improvement. This is where a testing suite is of importance for the success of the optimization.

4.4.1 Initial test

The initial tests were ran against the SQL solution and produced the results seen in Figure 16.

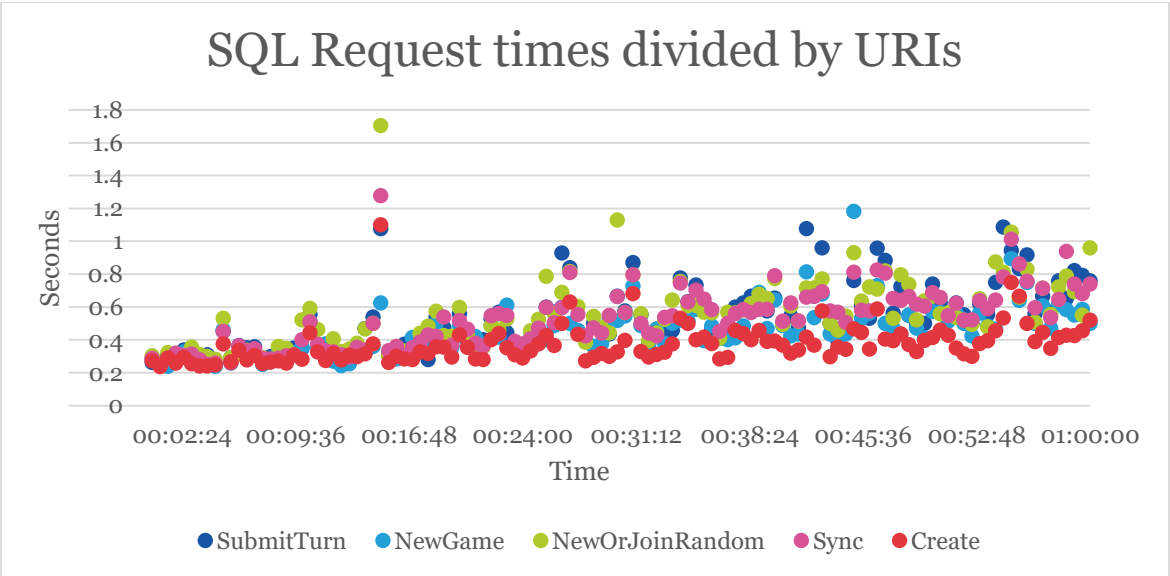

Figure 16 Request time between the different SQL URIs from the test without think times

4.4.2 Transactions

After careful evaluation and multiple testing iterations we discovered that transactional queries against a database, although adhering to ACID standards they brought with them considerable

response time and hindered throughput. After consideration we decided to not use transactions and instead rely on SQL's inherent atomicity.

4.4.3 Indexes

When looking at the test results from Figure 16 there is a clear trend in increasing request times as time go by. A hypothesis was that this trend was seen because of missing indexes. We then set out to improve the indexes both by analyzing our queries and add indexes for the most used fields, as well as using some automatic tools to find the indexes to add. With help from an industry professional we used a SQL query found in appendix 8.1 that tries to identify the missing indexes using SQL Servers Dynamic Management Views [39]. The identified indexes were then checked with the application to see that they were actually used extensively. All found indexes were and was therefore added as indexes. As can be seen in the final results in Figure 18 the increasing request times trend is gone.

4.4.4 Optimizing user Creation

When creating a new user, a lock was placed on the whole user table while looking for users with the chosen PlayerTag, putting a lock on the entire table is very bad for performance and could inflict the usability of the application as well as the response time. To get rid of this problem we used SQLs built in constraint that the PlayerTag is unique to update the table with a new entry. This got rid of the transaction and increased throughput, latency and increased the stability of user creation since no transactional lock needed to be created.

4.4.5 Optimizing Matching algorithm

The function for quickly finding a random game originally had a transaction with a range lock that locked everything the query read or affected, when this function reads a large majority of the current games table to compare with something else, that's a big problem. We quickly decided that this was a bad solution and after a while went with a solution that used some built in functionality of the SQL language to only update an entry if it was null. In that way we could be sure that we only inserted the user as defender if there were no defender assigned to it already.

4.4.6 Problems with the optimization

When using the built in SQL functionality to check that we only updated an entry that had defender set to null we had to sacrifice complete interoperability. All other database calls are built using the .NET entity framework and is normal C# code. This update that has the null check included can't be carried out using normal entity framework which means that it is stepped away from and instead a SQL-query is sent to the database as a string. This can be seen in Figure 17.

```
isUserNewDefender = db.Database.ExecuteSqlCommand(
    "UPDATE Homn.OnlineGames SET DefenderId = {0} WHERE Id = {1} AND DefenderId IS NULL",
    user.Id, existingGame.Id) == 1;
```

Figure 17 raw SQL query in C# code

4.4.7 Optimization results

When the final optimized solution had been created we ran a final test over 30 minutes using the natural test suite, producing Figure 18. And here we no longer see the response time increase which previously existed in Figure 16, we can also see a clear performance increase with the majority of the requests keeping below 0.3 seconds.

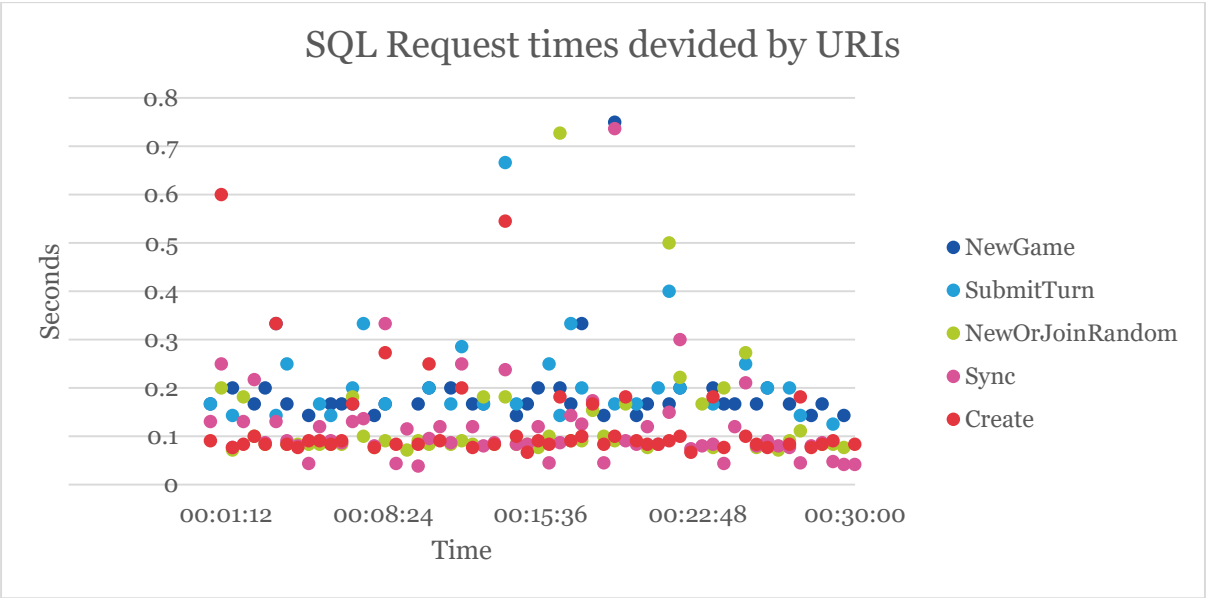

Figure 18 Request time between the different SQL URIs from the test with think times

4.5 Implementation of NoSQL

When starting to construct our implementation we began by simply creating pseudo code for the new implementation. This as to get a clear picture of what every method was supposed to do. Then we started to implement them.

4.5.1 Database design

The biggest change between relational and non-relational database is arguably how the databases are handled themselves. Since a non-relational database can have structures never possible in the relational world (such as multiple hierarchies) the first and most important part for performance and usability is the actual design of the database. The solution as seen in Figure 19 shows how we decided to save the data in the end. The three top classes is what is being saved directly to the database as documents. Then we can see that inside each NoMatchableGame there is one or more games. The fundamental difference here is that several users can have their own copy of the same game. If two players play against each other they both have a copy of the game. This makes some actions easier, for example to get all active games for a user we just get the “Games” property of that user. But trickier in other situations, for example because a game needs to be updated on two places when a turn is submitted.

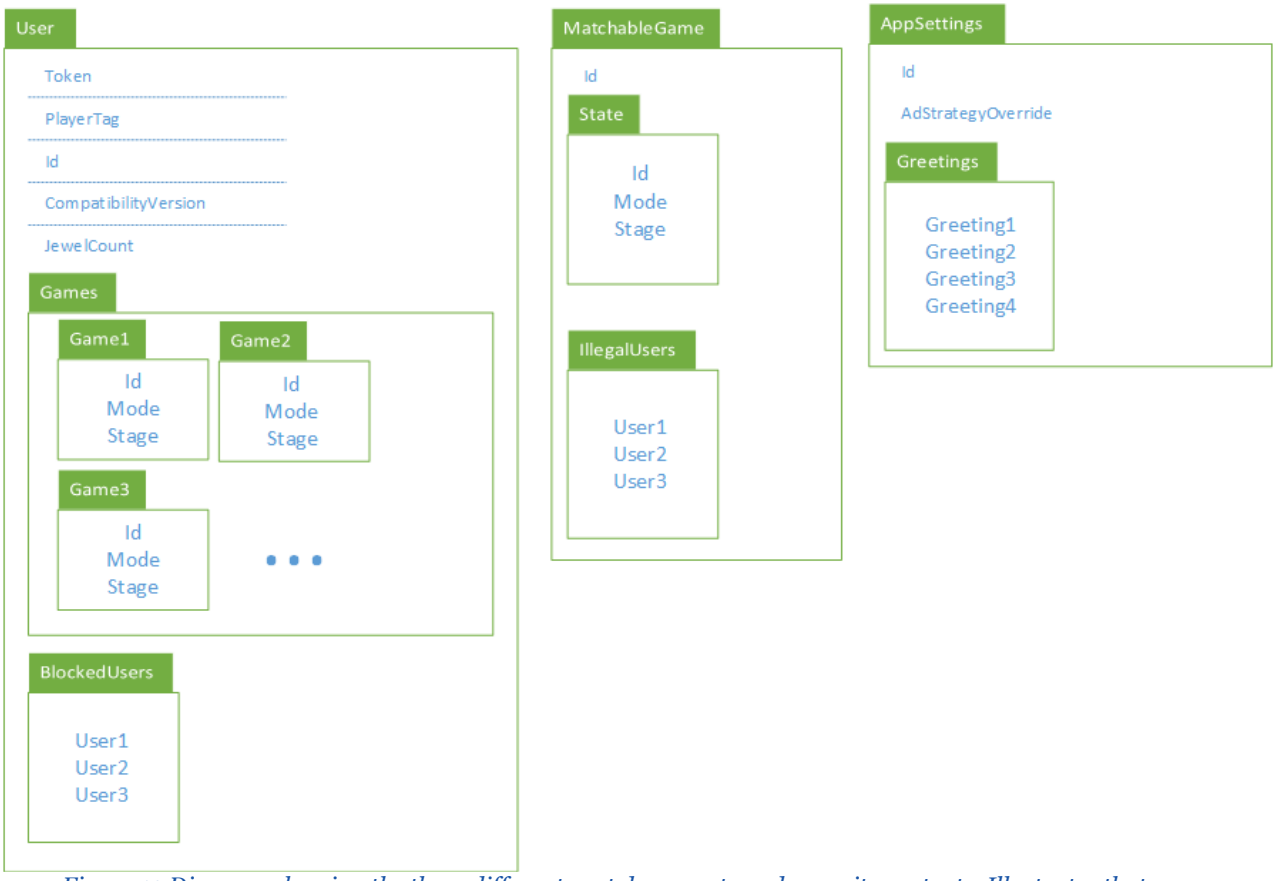

Figure 19 Diagram showing the three different root documents and some its contents. Illustrates that hierarchies are possible in Document stores.

4.5.2 Translation

First step was a simple translation from SQL to NoSQL. Very similar actions, same methods and the same flow as the SQL solution had. This created a working foundation that was used together with the tests to see that a NoSQL was at all possible with DocumentDB. This solution uses no special functionality provided by DocumentDB and has at times simply replaced the database call. As can be seen in Figure 20 this does not provide a good solution when running the tests. What happens is that just a few of the requests in the first 30 seconds succeeds until the RUs are used up and every proceeding call reaches timeout.

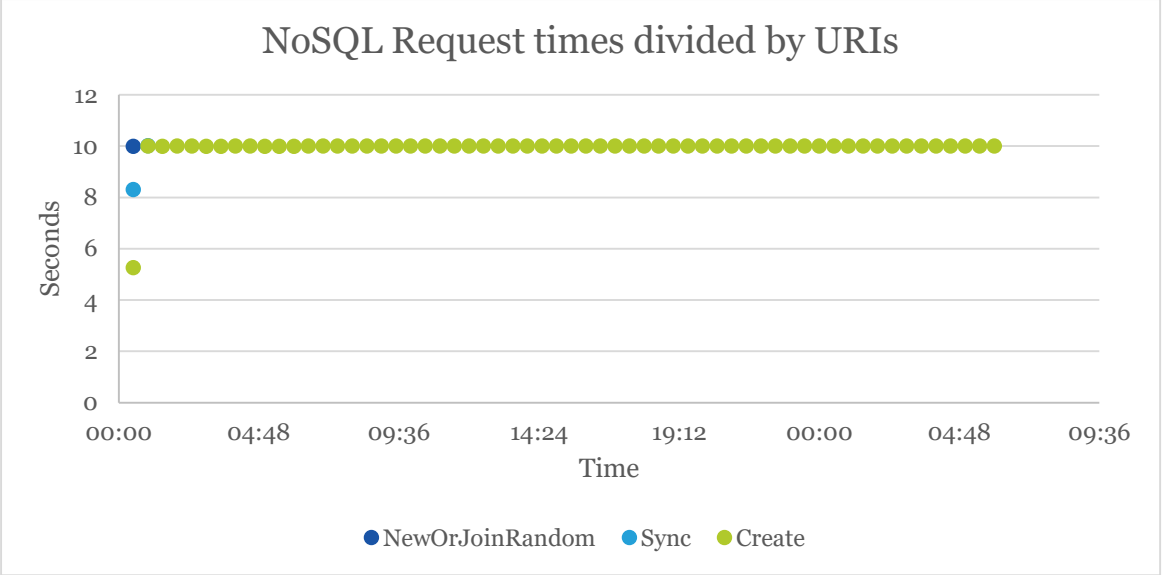

Figure 20 Request times between the different NoSQL URIs from the test without think times, here the time is constantly at 10 because that is when the timeout hits. Meaning that with six concurrent users constantly hammering the service in a load test, nothing really gets done

4.5.3 Optimization with Indexes

The standard indexing policy in DocumentDB is to index everything. The documents we are creating are either matchable games or users that contain games. Each game is 300+ lines of JSON data, and most of that data never needs to be queried by the database. Therefore a new index policy was created to ensure that instead of indexing 300+ data points to a maximum of 5 levels, the database only index 20+ data points to a maximum of 3 levels automatically [40].

The result was an increase from 1096 to 4843 tests in a fifteen minute period (visualized in Figure 21).

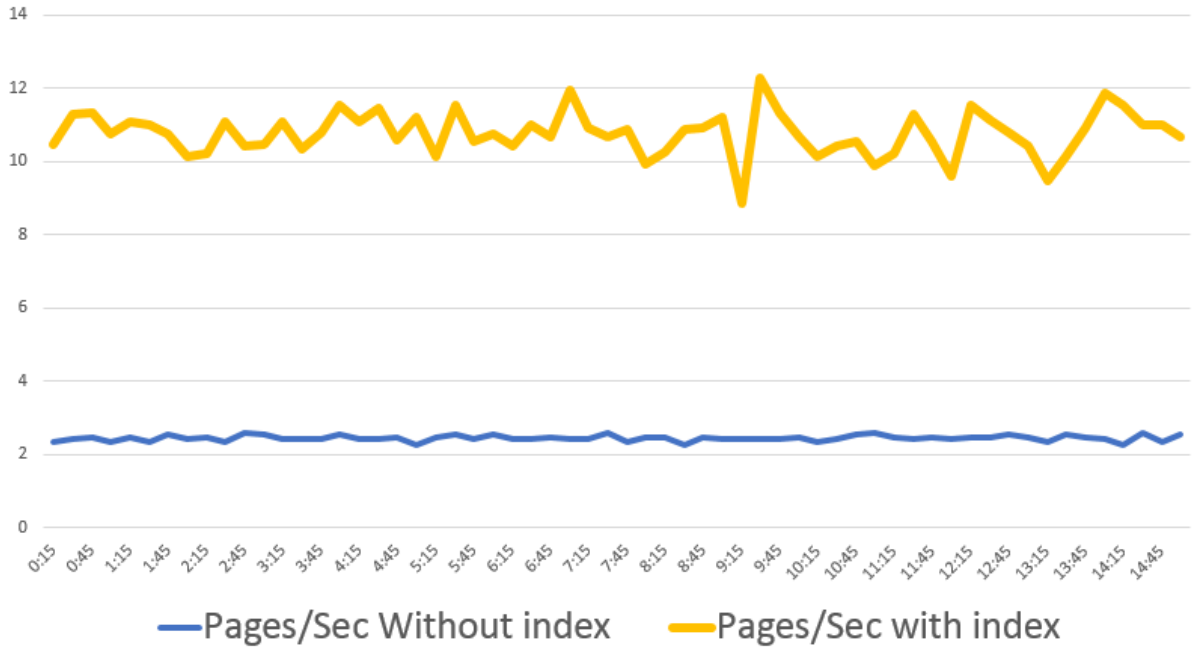

Figure 21 NoSQL solution tested with and without custom index

4.5.4 Optimization with stored procedures

The optimizations were done with regards from performance tips from the DocumentDB team in Redmond [41] [42] as well as from thoughts and ideas gathered during team discussions.

After adding indexes, the focus was on stored procedures. In relational databases stored procedures can be good for performance, but are mostly neglected because of Entity Framework that buffers up database changes and perform them all in a single call. In DocumentDB stored procedures are used in much the same way as Entity Framework is with SQL. Our stored procedures are mainly used to replace several calls to the database with one single call. This reduces the number of roundtrips to the server as well as introduce transactional security over multiple calls.

One good example of stored procedure is when submitting a turn. The game is located inside both users playing that game in the database. We want both games to be changed, and if one fails to change we want to rollback, so that we avoid inconsistency in the database, Figure 23 illustrates how the code looked before and after introducing a stored procedure in that scenario.

When running the natural test against the NoSQL implementation without stored procedures the data shown in Figure 22 was produced.

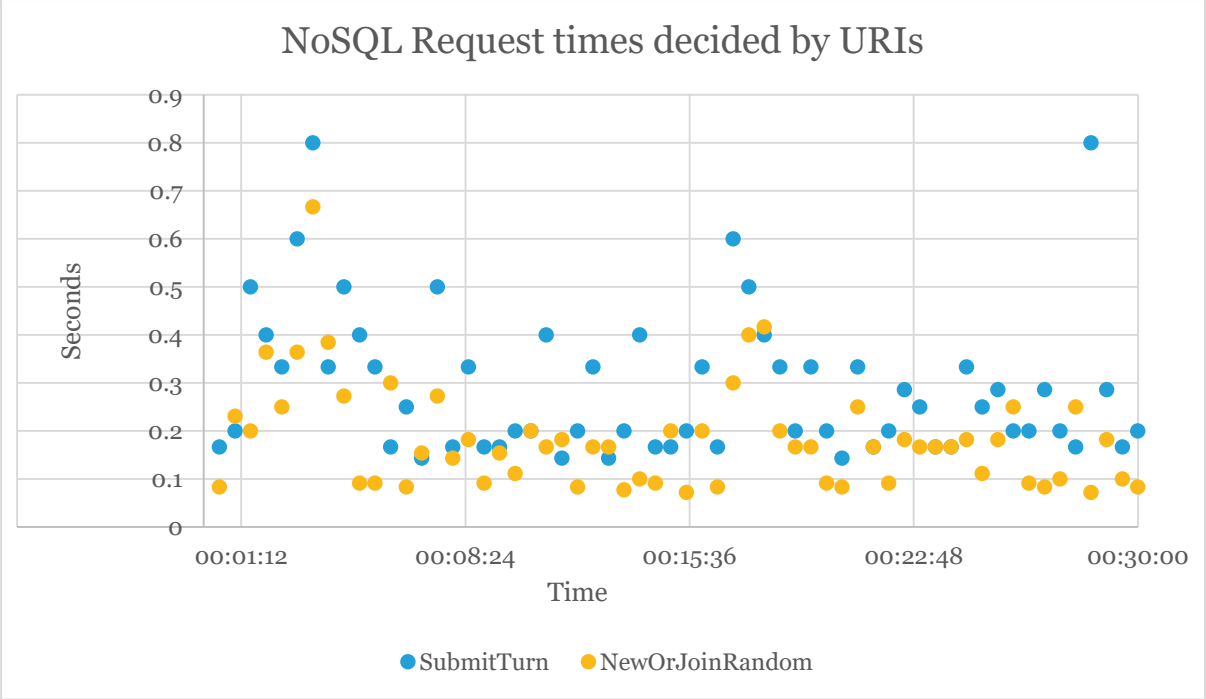

Figure 22 The test result from the NoSQL implementation without stored procedures (focusing on the requests that are effected by stored procedures)

The stored procedures are written using JavaScript and then sent to the server to be compiled into bytecode. The JavaScript code is therefore removed of the potential overhead it has. Stored procedures needs to be registered onto the collection before use, this is done the same way as you upload a document to the collection. Every stored procedure gets approximately 5 seconds of execution time, and will be notified prior to it being stopped by the return value in each CRUD (Create Read Update Delete) operation [43] [30]. JavaScript is really well suited for the DocumentDB database since it natively handles and processes JSON which is built upon it [44].

```
// OLD CODE
await ThrottledReplace(user);
await ThrottledReplace(opponent);

// NEW CODE
dynamic spResult = await CallDoubleUpdateSP(user, opponent);
```

Figure 23 Showing code for doing a multi-replace before and after introducing stored procedures

After optimizing we ran the tests using the natural scenario and then managed to produce the results seen in Figure 24. As can be seen, both request times have improved and the result seems to be a lot more reliable in time. The hypothesis for this is that when using stored procedures less RUs are used and therefore less throttling needs to be done which keeps the request time down.

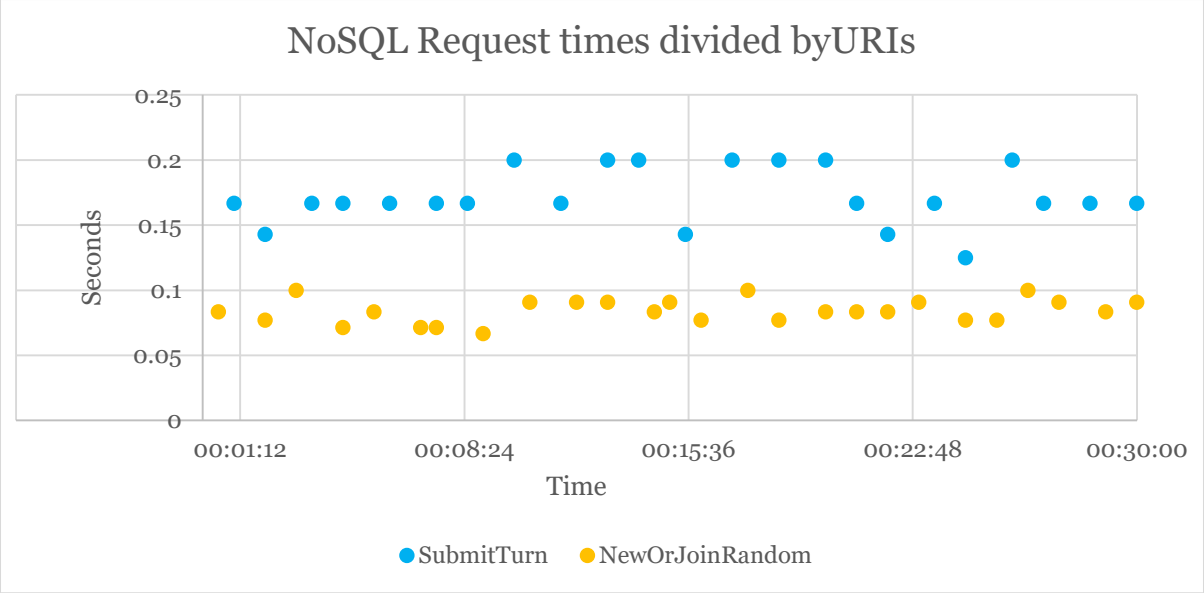

Figure 24 Request times between the different NoSQL URIs from the test with think times and stored procedures, only showing the requests affected by stored procedures

5 Conclusions

This chapter will outline our thoughts considering the analysis.

5.1 NoSQL vs SQL | Price & Scalability

DocumentDB natively supports scaling both vertically and horizontally, has no delay while scaling and can scale programmatically. These points makes this option natural when looking for a scalable data storage solution. Which is why we have come to the conclusion that DocumentDB is better in terms of scalability. Scalability is however closely related to price where the SQL price model is more developed, has more options and the cheapest tier is a lot cheaper than its DocumentDB counterpart. The flexibility and cost to get up and running quick with the Basic 0 tier makes this seem like the best choice price wise.

5.1.1 What are you paying for.

The two solutions uses different ways to bill the handling of data. Where Document DB's higher tiers yield an increase in Resource Units per second (RU/s). SQL higher tiers increase the computational power that is supplied by the server, measured in Data Transfer Units as well as storage.

5.1.2 How much are you paying.

As of writing this paper the prices for the cheapest tiers for DocumentDB and SQL are as given in the following tables:

Table 4. Excerpt from table 1 concerning the cheapest SQL pricing tier.

Tier	DTU	Size (GB)	Price / Hour (SEK)
Basic 0	5	2	0.045

Table 5. Excerpt from table 2 concerning the cheapest DocumentDB pricing tier.

Tier	RU / second	Size (GB)	Price / hour (SEK)
S1	250	10	0.229

It is easy to see that the cost for DocumentDB is roughly five times the cost of the SQL solution. You also get five times the space. As DTU's and RU's are very different it is very hard to judge performance based on this it is important to judge what sort of performance you need when evaluating the cost for the project.

Looking at Table 1 and Table 2 there is more flexibility in terms of the amount of tiers for SQL at the moment. Having a higher flexibility for the chosen solution in terms of pricing means there is a smaller chance of overspending when scaling out to accommodate higher loads.

When taking the ease to scale with DocumentDB and the ability to switch tiers for collections on the fly into account, the sometimes higher prices could be mitigated by switching tiers during times where loads are low and there's an excess of RU/s. Although a very clever strategy it might still not be enough to make a DocumentDB solution as cost effective as SQL.

This means that in terms of price and scalability SQL and DocumentDB are so different it is not the case that one is better but more expensive than the other. Instead it can only be described as that the NoSQL solution is better for datasets that are very dynamic in use and that need to change quickly with as little relations as possible. The SQL solution is better for datasets with many relations where the workload is without heavy fluctuations.

5.2 NoSQL vs SQL | .NET Interoperability

In our SQL solution there's only one piece of code that isn't written in .NET but as a regular SQL expression for a query, all other queries are done through .NET Entity Framework making this solution very easy to understand coming from a .NET Background. (code snippet in Figure 17)

Although the SQL solution is clearly more .Net interoperable, spending a couple of minutes getting to know the stored procedures will not prove a difficult task. Writing new ones on the other hand might be a bigger endeavor if you, the developer is not used to JavaScript.

By looking at Figure 13 it is quickly seen that both solutions got almost the exact same amount of .NET code, but that the NoSQL solution includes more non .NET code and other lines (comments or blank lines). This shows that the two different solutions require about the same amount of client code. But that NoSQL requires non .NET code in the server to perform optimally.

We therefore arrived at the conclusion that SQL Server is from a .NET interoperability standpoint better than DocumentDB.

5.3 NoSQL vs SQL | Performance

In the end our analysis shows that the NoSQL solution is faster than the SQL solution in all requests. As outlined in the analysis chapter this wasn't always the case. In the beginning the NoSQL solution was struggling to even handle very few concurrent users. After the optimization of the NoSQL solution the performance was improved multiple times over.

5.3.1 Know what you're using, and how it is supposed to be used

However, the NoSQL solution would still hit a wall and crumble under our performance tests. This was because of the fact that with the NoSQL solution you pay for the throughput per second. And not the reserved hardware capacity to use. This in turn led to that as soon as we hit that wall and requests got denied we effectively created an ever growing queue of requests with no opportunity for us to catch up. The SQL solution would instead of hitting a wall, simply slowdown. It took more time for each request to be performed if there were more clients using the server, and that was that.

5.3.2 Define the real problem, solve that

After the NoSQL solution had failed in our performance test we discovered that we were testing the wrong thing. We were not testing performance for the game, we tested the performance of the implementations during a stress-test scenario. A scenario that is never the case in the real world. After that we created a test that would simulate the real world, still with some extreme numbers but something that could happen in the real world. Now the NoSQL solution shines. NoSQL is not a new database that will take over and replace every relational database out there. It is a new database, for new situations and another set of problems that the relational database is not optimally designed for.

5.4 Final recommendation to Warmkitten

After drawing general conclusions in each of the three different areas, Price & Scalability, .NET Interoperability and Performance we had enough data and experience to make a recommendation to Warmkitten.

Because of the transactional nature of Warmkittens backend outlined in 2.2.3.3 and the limitations of DocumentDB explained in 2.4.2 that transactions can only be carried out in a single collection. We have decided to recommend Warmkitten to use the SQL solution.

DocumentDB is a new and exciting product that could bring considerate performance improvements as discovered in 4.3 as well as scalability flexibility as talked about in 4.1. But after our studies we've found out that Warmkittens game isn't one of those.

6 Discussion

6.1 Ethics & Sustainability

Comparing SQL Server with Document on the ethical notes comes down to the environment components of the two different solutions.

Ethically both solutions store and use the same kind of data. The data is also fully anonymized and never is a single email, name or other personal data stored or used. The only data that can possibly be connected to a player is the PlayerTag which the player chooses themselves.

Environmentally the solutions all reside on Microsoft data centers which have a strong dedication to create data centers not only fast and performant but also working towards a sustainable future [45]. In the environmental regard efficiency is very important, when creating SQL Server databases hardware is reserved for our application, during low-load times this means that there will be underutilized hardware waiting for something to happen, which isn't good for the environment. In the DocumentDB case the hardware is instead managed by Microsoft and the throughput is reserved, this enables a lot more efficient use of hardware and is better on the environment since less hardware is actually needed.

6.2 Unstable Decisions within Warmkitten

The company is owned and made up of one employee. All directional and prioritization decisions are made by Pontus Wittenmark. This makes the organization an extremely easy one from Warmkittens perspective.

The problem herein lies in the fact that a single person might change. Most decisions are not documented and the direction can be easily changed if Pontus wishes to do so, making the project somewhat volatile in its organization.

6.3 Future Work

This section will explore and talk lightly about those areas which we haven't touched upon that could be interesting for this study. It also talks about some studies that now might become interesting because of the results of this study.

6.3.1 Extended tests

The tests are engineered to test the critical parts for the SQL solution. It was engineered when only the SQL implementation existed and designed with that in mind. Those parts are not naturally the hardest for the NoSQL solution, something that leaves room for future explorations where SQL and NoSQL might differentiate themselves. Something else that would be nice to include in the tests are for example how many retries are needed for the NoSQL solution and the RU cost for each operation. And also if similar value could be extracted from the SQL server that would also be interesting to measure.

6.3.2 Other databases

This comparison has been made strictly between DocumentDB and SQL server. Both products created by Microsoft and consumed through the Microsoft service Azure. Something that could be interesting in the future is to create an extended test by adding more databases. Both relational and non-relational.

6.3.3 Different scenarios

Our scenario using the game developed by Warmkitten as a base makes some assumptions. Other scenarios might differ and our assumption may not apply to those scenarios.

6.3.4 More optimization

Our tests and implementation was heavily constrained by time. This means that we had to make decisions on where to optimize and how. With more time more optimization could be done in both fields and for example more stored procedures could be designed and implemented.

6.3.5 Elastic Databases

Elastic databases was very recently announced and could give another conclusion if that was to be used instead [46].

6.3.6 Compare RUs and DTUs more in-depth

Our comparison simply scratches the surface and mostly uses data easily obtainable. We did have a interview planned with the people that created the product. But unfortunately time-zone troubles stopped us from managing to have the interview before the publishing of this paper. Carrying out that interview and doing a more thorough comparison would be good for this paper.

6.4 Validity of methods and data

The methods used have been selected together with senior software developers. It was chosen because of our familiarity with the iterative process and that we had used it before. In retrospective we can conclude that it seems to have worked well in collaboration with DocumentDB and that the method never was in our way of accomplishing our task at hand. The iterative process enabled us to quickly make changes, something that was vital when developing at the same time as we were gaining knowledge of the product used. Other methods may be as good or better with this task but that is nothing that we evaluated.

The data gathered has been done in such a way as to minimize outside factors. This includes running the tests as close to the tested hardware as possible. A big factor is that we cannot effect the actual hosting. Microsoft owns and manages the hardware used and there is a possibility that they changed something on their end without us knowing about it. Such a change could have influenced our results, but is considered unlikely. The test suite and tools used could influence the results produced, this would however most likely produce the same effect on both implementations. Therefore, we trust our data points to be correct relatively to both implementations.

6.5 Our work in perspective

This thesis answers Warmkittens questions about whether or not a NoSQL solution using DocumentDB is better for them than their current SQL solution. Using this as a case study, the conclusions we have reached can also be used for other companies in similar situations. It is however important to note that this case study is highly specific, which implies that any differences from Warmkittens case could create another conclusion than what is presented in this paper.

One such change could be the complexity of the database. The SQL database consisted of less then ten tables, which can be considered to be not very complex. A more complex database could produce different results in the comparison.

7 References

[1] L. O. E. J. A. K. Noel Yuhanna, "NoSQL Key-Value Databases, Q3 2014," 2014. [Online]. Available: https://www.forrester.com/The+Forrester+Wave+NoSQL+KeyValue+Databases+Q3+2014/fulltext/-/E-RES118782.

[2] M. Fowler, "NosqlDefinition," 9 January 2012. [Online]. Available: http://martinfowler.com/bliki/NosqlDefinition.html.

[3] M. Rouse, "NoSQL (Not Only SQL database)," October 2011. [Online]. Available: http://searchdatamanagement.techtarget.com/definition/NoSQL-Not-Only-SQL.

[4] G. Kumar, 3Pillar Global, 2014. [Online]. Available: http://www.3pillarglobal.com/insights/exploring-the-different-types-of-nosql-databases. [Accessed 09 April 2015].

[5] M. Seeger, "Key-Value stores: A practical overview," medien informatik, 2009.

[6] R. CrawCour, *Developing Solutions with Azure DocumentDB,* 2015.

[7] Microsoft, "DocumentDB - NoSQL data management | Microsoft Azure," 2015. [Online]. Available: https://azure.microsoft.com/en-us/services/documentdb/. [Accessed 28 March 2015].

[8] Solid IT, "DB-Engines Ranking - popularity ranking of database management systems," Solid IT, April 2015. [Online]. Available: http://db-engines.com/en/ranking. [Accessed 28 March 2015].

[9] RJMetrics, "Query Translator," 17 April 2015. [Online]. Available: http://www.querymongo.com/.

[10] D. Valz, "Rules of Engagement - NoSQL Column Data Stores," 28 February 2013. [Online]. Available: http://www.ingenioussql.com/2013/02/28/rules-of-engagement-nosql-column-data-stores/. [Accessed 09 April 2015].

[11] I. Robinson, J. Webber and E. Eifrem, Graph Databases, O'Reilly Media, 2013, pp. vii-6.

[12] Facebook Inc, "Graph API Overview," Facebook, [Online]. Available: https://developers.facebook.com/docs/graph-api/overview. [Accessed 09 April 2015].

[13] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," 1970.

[14] J. C. McCallum, "Disk Drive Prices (1955-2014)," 14 September 2014. [Online]. Available: http://www.jcmit.com/diskprice.htm. [Accessed 28 March 2015].

[15] Amazon, "Amazon.com: WD Green 2TB Desktop Hard Drive: 3.5-inch, SATA 6 Gb/s, IntelliPower, 64MB Cache WD20EZRX: Computers & Accessories," Amazon, 16 September 2013. [Online]. Available: http://www.amazon.com/Green-2TB-Desktop-Hard-Drive/dp/B008YAHW6I/. [Accessed 28 March 2015].

[16] CoinNews, "Inflation Calculator | Find US Dollar's Value from 1913-2015," CoinNews Media Group, LLC , 24 March 2015. [Online]. Available: http://www.usinflationcalculator.com/. [Accessed 3 April 2015].

[17] P. Carbonnelle, "TOPDB Top Database index," 2015. [Online]. Available: http://pypl.github.io/DB.html. [Accessed 28 March 2015].

[18] *Oren Eini Does NoSQL First*. [Sound Recording]. .NET Rocks!. 2012.

[19] S. Sakr, "Supply cloud-level data scalability with NoSQL databases," 2013.

[20] Microsoft, "Description of the database normalization basics," 12 July 2013. [Online]. Available: https://support.microsoft.com/en-us/kb/283878?wa=wsignin1.0. [Accessed 28 March 2015].

[21] R. CrawCour, Interviewee, *Dev.Cast 78 – DocumentDB, the new kid on the nosql block*. [Interview]. 15 September 2014.

[22] MongoDB, "A Database that Moves at Your Speed | MongoDB," MongoDB, 2015. [Online]. Available: http://www.mongodb.com/industries/high-tech. [Accessed 28 March 2015].

[23] P. Wittenmark, "Warm Kitten," 2013. [Online]. Available: http://www.warmkitten.com. [Accessed 28 March 2015].

[24] Microsoft, "Pay-As-You-Go," [Online]. Available: http://azure.microsoft.com/en-

us/offers/ms-azr-0003p/. [Accessed 15 May 2015].

[25] Microsoft, "SQL Database Pricing," [Online]. Available: http://azure.microsoft.com/en-us/pricing/details/sql-database/. [Accessed 15 May 2015].

[26] Microsoft, "DocumentDB Pricing," [Online]. Available: http://azure.microsoft.com/en-us/pricing/details/documentdb/. [Accessed 15 May 2015].

[27] A. Liu, "Scaling a Multi-Tenant Application with Azure DocumentDB," 3 December 2014. [Online]. Available: http://blogs.msdn.com/b/documentdb/archive/2014/12/03/scaling-a-multi-tenant-application-with-azure-documentdb.aspx.

[28] S. Higa, "Azure SQL Database elastic database tools topics," 4 April 2015. [Online]. Available: http://azure.microsoft.com/en-us/documentation/articles/sql-database-elastic-scale-documentation-map/.

[29] J. Macintyre, "Performance levels in DocumentDB," 14 April 2015. [Online]. Available: http://azure.microsoft.com/en-us/documentation/articles/documentdb-performance-levels/#changing-performance-levels-using-the-azure-preview-portal.

[30] M. Gentz, "DocumentDB limits and quotas," 05 April 2015. [Online]. Available: http://azure.microsoft.com/en-us/documentation/articles/documentdb-limits/.

[31] Microsoft, "Generate and run a coded web performance test," 19 May 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms182552.aspx.

[32] Microsoft, "Testing Performance and Stress Using Visual Studio Web Performance and Load Tests," [Online]. Available: https://msdn.microsoft.com/en-us/library/dd293540(v=vs.110). [Accessed 19 May 2015].

[33] D. Chappel, "Introducing DocumentDB - A NoSQL Database for Microsoft Azure," Chappel & Associates, 2014.

[34] A. Schram and K. M. Anderson, "MySQL to NoSQL: data modeling challenges in supporting scalability," in *SPLASH '12 Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, New York, 2012.

[35] M. Stonebraker, "SQL databases v. NoSQL databases.," *Communications of the ACM,* vol. 53, no. 4, pp. 10-11, April 2010.

[36] Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases," in *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, Victoria, BC, 2013.

[37] J. Koch, "Developing Solutions with Azure DocumentDB," 16 April 2015. [Online]. Available: http://channel9.msdn.com/Series/Developing-Solutions-with-Azure-DocumentDB.

[38] M. Gentz, "Manage DocumentDB capacity needs," 29 April 2015. [Online]. Available: https://azure.microsoft.com/en-us/documentation/articles/documentdb-manage/.

[39] A. K. Mehta, "Identify Missing Indexes Using SQL Server DMVs," [Online]. Available: http://www.sql-server-performance.com/2009/identify-missing-indexes-using-sql-server-dmvs/. [Accessed 28 May 2015].

[40] F. Miiro, "Is it worth using custom indexing in DocumentDB?," 16 May 2015. [Online]. Available: http://fabianmiiro.se/the-impact-of-indexing-in-documentdb/.

[41] S. Baron, "Performance Tips for Azure DocumentDB – Part 2," 27 January 2015. [Online]. Available: http://azure.microsoft.com/blog/2015/01/27/performance-tips-for-azure-documentdb-part-2/.

[42] M. Gentz, "DocumentDB server-side programming: Stored procedures, triggers, and UDFs," 6 May 2015. [Online]. Available: http://azure.microsoft.com/en-us/documentation/articles/documentdb-programming/#stored-procedures.

[43] Microsoft, "DocumentDB programming: Stored procedures, triggers, and UDFs," [Online]. Available: http://www.documentdb.com/javascript/tutorial. [Accessed 21 May 2015].

[44] D. Crockford, "Introducing JSON," [Online]. Available: http://json.org/. [Accessed 21 May 2015].

[45] Global Foundation Services, Microsoft, "Datacenter Sustainability," [Online]. Available: http://www.microsoft.com/en-us/server-cloud/cloud-os/global-datacenters.aspx#Fragment_Scenario6. [Accessed 21 May 2015].

[46] S. Higa, "Azure SQL Database - elastic database tools," Microsoft Azure, 24 April 2015. [Online]. Available: http://azure.microsoft.com/en-us/documentation/articles/sql-database-elastic-scale-introduction/. [Accessed 26 May 2015].

8 Appendix

8.1 Finding missing indexes query

```
SELECT TOP 20
ROUND(s.avg_total_user_cost * s.avg_user_impact * (s.user_seeks + s.user_scans),0) AS [Total
Cost] ,
d.[statement] AS [Table Name] ,
equality_columns ,
inequality_columns ,
included_columns
FROM sys.dm_db_missing_index_groups g
INNER JOIN sys.dm_db_missing_index_group_stats s
                ON s.group_handle = g.index_group_handle
INNER JOIN sys.dm_db_missing_index_details d
                ON d.index_handle = g.index_handle
ORDER BY [Total Cost] DESC
```

8.2 Regex to find code lines

```
^(?([^\r\n])\s)*[^\s+?/]+[^\n]*$
```

TRITA-ICT-EX-2015:171