Examensarbete

HÖGSKOLAN I SKÖVDE

JÄMFÖRELSE MELLAN MYSQL OCH NoSQL VID LAGRING AV ANVÄNDNINGSDATA I MICROSERVICE-APPLIKATION/ARKITEKTUR

COMPARISON BETWEEN MYSQL AND NoSQL WHEN STORING USAGE DATA IN A MICROSERVICE ARCHITECTURE

Examensarbete inom huvudområdet Informationsteknologi
Grundnivå 30 högskolepoäng
Vårtermin 2021

Mathias Naess

Handledare: Henrik Gustavsson
Examinator: Mikael Berndtsson

Sammanfattning

Microservice-arkitektur är en växande trend inom utvecklingen av applikationer, även lagring av användningsdata. Lagringen av användningsdata sker bland annat i syfte för att öka *Quality of service* vilket innefattar bland annat tiden vid sidladdning. I rapporten utförs experiment för att mäta påverkan lagringen av användningsdata har på sidladdningstiden och huruvida den påverkan går att mitigera genom optimering av lagring i form av val av databassystem. En Microservice applikation utvecklas med alternativa lagringsstrategier för MongoDB och MySQL vad gäller lagring för användningsdata. Lagringen av användningsdata sker enligt web mining tekniken *Web Usage Mining* och mätningar utförs på laddningstiden med hjälp av automatiserade användare genom användningen av javascript. Resultaten av mätningarna analyseras sedan för att avgöra vilken databashanterare som är mest lämpad för lagring av användningsdata i en applikation byggd i microservice-arkitektur.

Innehållsförteckning

1. Introduktion

Bland webbapplikationer i dagens samhälle tenderar migrationen från användandet av monolitisk arkitektur till microservice arkitektur, och flera jättar såsom Netflix, Amazon och Ebay migrerat över(Al-Debagy och Martinek, 2018).
Idag genereras även en massiv volym användningsdata av användarna, oavsett arkitekturen applikationen är byggd med. För att använda sig av den genererade datan för att exempelvis tolka och registrera mönster i syfte att förbättra *Quality of service* kan olika tekniker utnyttjas, bland annat Web Usage Mining.
Quality of service är nyckeln att bedöma hur väl webbaserade applikationer uppfyller kundernas förväntningar på två primära åtgärder: tillgänglighet och svarstid (Menasce, 2002). Svarstiden vid sidladdning kan påverkas av flera olika faktorer, varav en faktor som kan påverka är lagringssystemet tjänsten använder, för att lagra samt hämta data.

När denna typ av lagring av användningsdata ska utföras finns det flera olika databassystem att välja bland, från traditionella relationsdatabaser så som MySQL och MariaDB till nyare NoSQL databaser. Därför är det viktigt att utvecklaren använder sig av rätt databassystem för att tillåta så bra quality of service som möjligt, speciellt då lagringen av användningsdata bland annat sker i syfte att förbättra användarens upplevelse.

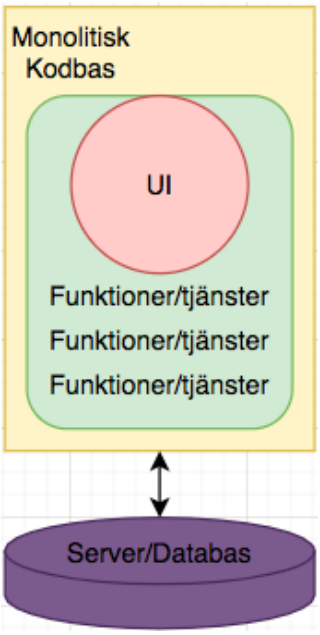
I arbetet kommer en applikation utvecklas baserad på microservice-arkitektur vars syfte är att lagra användarens handlingar med hjälp av olika databassystem. Ett tekniskt experiment kommer att utföras i vilket sidladdning mäts. Mätresultaten kommer sedan utvärderas för att avgöra påverkan valet av databassystem har vid lagring av användningsdata hos en microservice applikation.

2. Bakgrund

Detta arbetet fokuserar på hur valet av databassystem påverkar svarstiden på hemsidor byggda med en microservice arkitektur vid lagring av användningsdata. De tekniker som kommer utnyttjas i arbetet innefattar HTML, Javascript, Php, MySQL, MongoDB och Web Usage Mining. I en tidigare studie jämför JMTauro, Aravindh och Shreeharsha (2012) olika NoSQL databaser och nämner bland annat att allt fler projekt innefattande en större mängd data överväger att använda sig av MongoDB istället för en traditionell relationsdatabas.
Eftersom valet av databassystem och hur det påverkar responstiden vid sidladdning hos webbtjänster vid loggning av användningsdata används tekniken Web Usage Mining för att utvinna den relevanta data en användare genererar vid användandet av olika webbtjänster. Reddy, Reddy och Sitaramulu (2013) beskriver hur man effektivt utför förbehandlingen av data genererat i web access loggen, vilket är det första och mest relevanta steget inom web usage mining för det här projektet.

2.1 Mjukvaruarkitekturer

Flera forskare så som Damodaran, Salim, och Vargese, (2016) och Parker, Poe och Vrbsky (2013) m.fl. har i tidigare studier jämfört prestandan hos system som använder sig av relationsdatabassystem och så kallade NoSQL-system. I dessa studier är applikationerna baserade på en traditionell monolitisk arkitektur, vilket eventuellt riskerar att bli föråldrat då allt fler tjänster så som netflix och amazon byter till den trendande microservice arkitekturen(Al-Debagy och Martinek, 2018). Därav utförs denna studien vars syfte är jämföra dessa olika lagringsstrategier i en applikation baserad på en modernare arkitektur. Exempel på en monolitisk arkitektur samt kodexempel för utskrivning av data hämtat från en databas syns i figur 1 och 2.



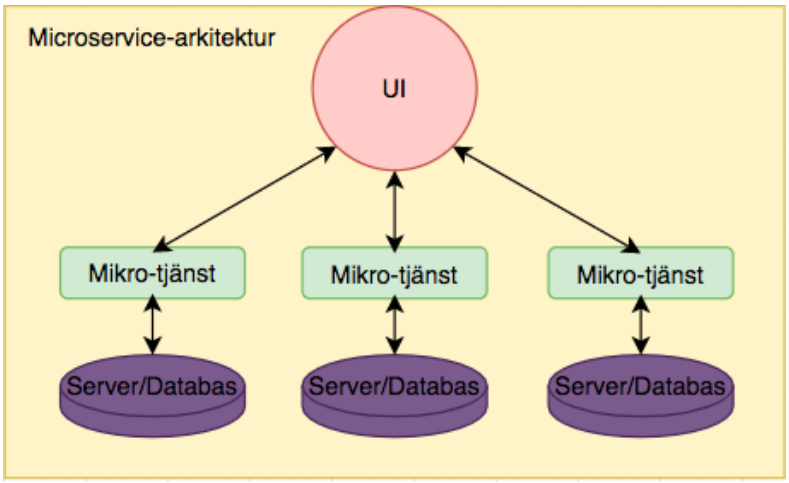
Figur 1: Exempel Monolitisk arkitektur

```
Applikation.php

Select * from kund
echo "<table>";
echo "<tr>";
echo "<td> kund ID </td>" ;
[...]
Select * from konto
echo "<div>";
echo "<h1> kontosida </h1>";
[...]
```

Figur 2: Monolit kodexempel för kund- och kontotjänster

Al-Debagy och Martinek (2018) diskuterar, i artikeln *A Comparative Review of Microservices and Monolithic Architectures,* användandet av applikationer baserade på de olika arkitektur-typerna microservices och monolitisk arkitektur. De utför diverse prestandamätningar samt diskuterar för och nackdelar innefattande bland annat skalbarheten ett system byggt med microservice arkitektur medför, vilket är försumbart i förhållande till den här rapporten.



Figur 3: Exempel Mikroservice-arkitektur

```
kundtjänst.php
<?php>
[...]
<?>
<html>
<div id='kundlist'>
</div>
[...]
```

Figur 4: Kundtjänst microservice exempel

```
kontotjänst.php
<?php>
[...]
<?>
<html>
<div id='kontodiv'>
</div>
[...]
```

Figur 5: Kontotjänst microservice exempel

Ännu en fördel, vilket är försumbart i detta fallet med microservices, är att det underlättar för fler utvecklar-team att arbeta på systemet samtidigt då varje microservice är en självständig del i systemet, se figur 3.
Trots att fler större företag så som *Netflix, Amazon* och *Ebay* har migrerat deras tjänster till en microservice arkitektur (Al-Debagy & Martinek, 2018) innebär inte det att det nödvändigtvis är bättre vad gäller responstid vid sidladdning, eftersom förfrågningar måste skickas genom porten till alla microservices tjänsten använder.
Skillnaden mellan monolitisk och microserver arkitektur, är att den monolitiska arkitekturen består av en samlad kodbas som inkapslar tjänstens alla funktioner. Mindre projekt byggda med en monolitisk arkitektur har sina fördelar; de är enkla att utveckla, testa och sätta i drift (Chen, Li & Li, 2017).
I och med att applikationen växer medför den samlade kodbasen dock komplikationer, exempelvis kan kodbasen bli överväldigande komplex vilket innebär att bug-fixning och tilläggning av nya funktioner blir svårhanterat. Ytterligare ett problem är att om en komponent går ner påverkas hela systemet (Chen, Li & Li, 2017).
I en microservice-arkitektur exekveras varje komponent individuellt, vilket innebär att de olika funktionerna inte är beroende av varandra för att köras, se figur 4 & 5. Det innebär även att man kan utnyttja olika tekniker i de olika mikro-tjänsterna efter behov (Al-Debagy och Martinek, 2018).

2.2 MongoDB

Braun et al.(2017) skriver att de olika mikrotjänsterna i en microservice arkitektur kan innefatta ett internt databassystem, men att databassystemen vanligtvis är separerade från mikrotjänsterna för att upprätthålla statlöshet i tjänsterna. De beskriver även att system som hanterar *Big Data* kan utnyttja verktyg så som NoSQL system integrerade i microservice arkitekturen. Denna integrerade data hanterings arkitektur består av olika mikrotjänster vilket kan inkludera följande :

- Link-service, som skapar relationer mellan data och länkar samman olika data-objekt.
- Schema-service, vars uppgift det är att hantera schemabeskrivning för de olika data-objekten lagrade i de olika tjänsterna, så som format och datatyper.
- Master Data-service, som hanterar strukturerad data.

Kommunikationen vad gäller bland annat formateringen av data mellan de olika mikro tjänsterna sker i största del i form av json vilket innebär att ett databassystem som använder sig av Json struktur är fördelaktigt för att lagra schemabeskrivningarna (Braun et al., 2007).

NoSQL är den breda termen för lagring av data som inte använder sig av relationsdatabaser, de använder sig av key/value lagring lämpade för de tillfällen då hastighet vid infogning, läsning och skrivning är nödvändigt(Gu, Wang, Shen, Wang, & Kim. 2015). MongoDB är en av de populäraste NoSQL databaserna, vars uppgift det är att överbrygga klyftan mellan traditionella relationsdatabaser och key/value databaser genom att utnyttja fördelarna hos båda(Gu et al., 2015).
MongoDB använder sig av dokument-baserad lagring som använder sig av BJSON data strukturer som stödjer komplexa datatyper. Det använder sig även av ett kraftfullt query språk och har stöd för indexering(JMTauro, Aravindh & Shreeharsha, 2012). Exempel av insert, query och objekt syns i figurerna 6, 7 & 8.

```
var embeddedObject={"gata":"Skövdegatan","stad":"Skövde"}
Db.anvandare.insert({"id": 1, "namn": "Mathias",
"adress":embeddedObject})
```

Figur 6: Insert i MongoDB

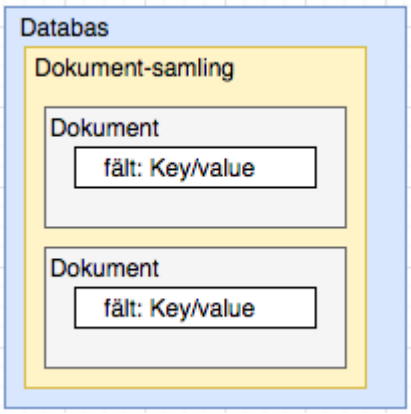
```
{"id":1,
"namn": "Mathias",
"adress":{
"gata":"Skövdegatan",
 "stad":"Skövde"}}
```

Figur 7: Exempel objekt av en användare i ett MongoDB system

```
 db.anvandare.find({"id":1, "namn":"Mathias"})
```

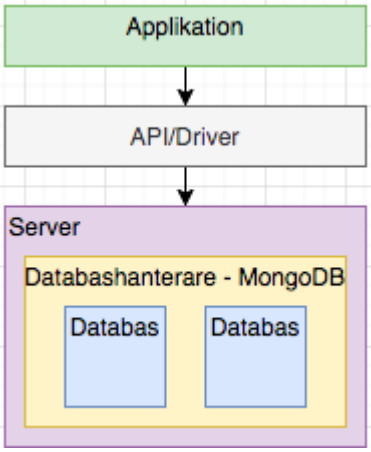
Figur 8: Exempel MongoDB Json query för att hitta denna användare

Databasen innehåller en samling av dokument som består av key/value fält. Key fältet är namnet av fältet och förekommer i string datatypen och value-fältet är datan som lagrad och kan förekomma i vilken datatyp som helst (Gu et al. 2015). Strukturen går att se i figur 9.



Figur 9: MongoDB databas struktur

Lagringsarkitekturen består av att applikationen skickar requests till den valda API/Drivern som sedan kommunicerar vidare till databashanteraren på servern vilket i sin tur lagrar datan i databasen(Gu et al 2015). Se figur 10.



Figur 10: Arkitektur vid lagring av data

Lawrence (2014) Diskuterar i artikeln *Integration and Virtualization of Relational SQL and NoSQL Systems Including MySQL and MongoDB* användandet av *The Unity virtualization driver* vars uppgift är att parsa, översätta och optimera queries skrivna med SQL syntax till olika NoSQL varianter så som MongoDB. det förekommer dock en overhead vilket innebär fördröjning på hastigheten av resultaten jämfört med Mongo Java API

Parker, Poe och Vrbsky (2013) 'Comparing NoSQL MongoDB to an SQL DB'
använder sig i sin studie av den senaste versionen av MongoDB C# drivers vid jämförelse av mongo och sql, dock måste vi ta hänsyn till att 7 år passerat sen dess och nya lämpligare alternativ kan finnas till hands.

2.3 Web Usage Mining

Behovet av flexibilitet vid lagring och åtkomst av data banade väg för dokument-baserade databassystem på grund av de dokument-baserade databassystemens förmåga att hantera stora mängder komplex data i olika sturkturer. Trots skalbarheten hos lagringssystemen bidrar storleken av web access loggarna till svårigheter vid extraktion av användares beteéndemönster(Meenatchi et al., 2018). På grund av det används data-mining tekniker för att sammanfatta den relevanta informationen lagrad i web access loggen, utöver sammanfattningen utförs det även formatering av datan vilket är fördelaktigt vid bearbetningen(Meenatchi et al., 2018). I artikeln *A New NoSQL Framework for Effective Interpretation in Web Usage Mining(2018)* innehåller den råa web log filen 232 000 data-entries, vilket de med hjälp av Web Usage Mining tekniker skalar ner till 141 000 och därmed minskar volymen av datan lagrad.
Inbarani & Thangavel (2006) förklarar att behovet för att utveckla nya tekniker vars syfte det är att förbättra bland annat responstid och användarupplevelser ökar i samband med den volym data som genereras på internet. Li och Feng (2008) förklarar att tjänsteleverantörerna med hjälp av tekniken web usage mining kan förbättra sina webbsidor, genom att analysera hur webbsidorna används och på så vis upptäcka användarnas intressen och deras åtkomstmönster. Dvs. hur de navigerar sig på webbsidan för att få åtkomst till det de är ute

efter. Data mining innebär att man med hjälp av data-driven teknik upptäcker mönster i stora volymer av data. Web mining uppnås genom tillämpningen av dessa data mining tekniker på webb data(Reddy, Reddy & Sitaramulu 2013). Web usage mining består oftast av tre huvudsakliga steg(Li & Feng 2008):

1. Förbehandling av data
2. Kunskapsutvinning
3. resultatanalys

Steg 1, förbehandling av data, är den som kommer vara mest relevant för det här projektet.
Förbehandlingen av data innefattar olika faser: *Data Cleaning, User identification, Session identification* och *Path completion(*Reddy, Reddy & Sitaramulu, 2013)
User identification och Session identification kan sammanfogas till en och samma fas och då innebära *User & Session Identification (*Li & Feng 2008). Men slutresultatet blir detsamma.
Data Cleaning fasen innebär att man städar upp datan och gör sig av med irrelevant och överflödig data, då syftet med web usage mining är att skapa sig en klar bild av webbanvändares beteende (Reddy, Reddy & Sitaramulu, 2013).
Fasen *User & Session identification* går ut på att skilja på olika användarsessioner i den ursprungliga web access loggen (Li & Feng, 2008).
Path completion fasens syfte är att färdigställa inkompletta åtkomst mönster då det är många sökvägar som inte sparas i web access loggen på grund av bland annat lokal cachning(Li & Feng, 2008).

Förbehandlad data		
User:ID	Session:ID	User Path/clickstream
Exempeldata:		
Ipadress	date-time	Url.1-Url.2-Url.3 etc

Figur 11: Exempel på datastruktur av förbehandlad data

3. Problemformulering

Reddy, Reddy & Sitaramulu (2013) förklarar att Web Usage Mining tekniken kan tillämpas för hantera den massiva volymen data som genereras i en evigt växande World Wide Web. De beskriver att produkten av Web Usage Mining kan användas för att förbättra systemets responstid, modifiering av webbplatsen, business intelligence, användarkarakterisering etc.
Menasce (2002) beskriver *Quality of service* som nyckeln att bedöma hur väl webbaserade applikationer uppfyller kundernas förväntningar på två primära åtgärder: tillgänglighet och svarstid. Och att dålig Quality of service i form av lång svarstid leder till frustrerade användare, vilket i sin tur kan leda till förlorade affärsmöjligheter.
Tidigare studier så som studien utförd av Al-Debagy och Martinek (2018) som utfört prestanda-utvärderingar i samband med byte av databassystem från relationsdatabas till NoSQL har genomförts på system byggda med en monolitisk arkitektur. På grund av det riskerar de att mista sin relevans i en industri i vilken migration av applikationer från monolitisk- till mikroservicearkitektur är trenden då fler stora företag så som Netflix, Amazon och Ebay flyttat över till microservice-arkitekturer(Al-Debagy and Martinek, 2018). På grund av denna brist av forskning vet vi inte hur de olika databassystemen förhåller sig till varandra vad gäller svarstider i en modernare arkitektur.
I studien som genomfördes av Damodaran, Salim & Vargese (2016) indikerar resultaten att i en applikation för stormarknader, som hanterar ett stort utbud av produkter vilket innebär en omfattande mängd data, var svarstiderna vid inserts marginellt bättre vid användandet av MongoDB jämfört med MySQL. Vilket inte nödvändigtvis stämmer om vi ändrar arkitekturen på applikationen dessa mätningar utförs i.

Frågeställning: Påverkar valet av databassystem vid lagring av användningsdata enligt web usage mining tekniken svarstiderna hos en applikation byggd med microservice arkitektur

Hypotesen är att valet av databassystem kommer att påverka responstiden hos webbsidan.
Nollhypotesen är att responstiden hos webbsidan är oberoende av databassystem.
Delhypotes1: Responstiden kommer vara effektivare vid användande av MongoDB som databassystem.
Delhypotes2: Responstiden kommer vara effektivare vid användande av MySQL som databassystem.

Det kan även vara relevant att undersöka hur systemets svarstid påverkas av datainsamlingen som sker i förbättringssyfte.

4. Metod

4.1 Metodbeskrivning

Den valda metoden för det här projektet är experiment, det kommer utföras empiriska mätningar i form av prestanda mätningar vad gäller svarstiden på webbsidan vid lagring av användningsdata. Användande data innebär den data som genereras av en användare då användaren utför handlingar, dvs. exempelvis de klick en användare utför. Den primära datainsamlingsmetoden för kvalitativ och kvantitativ data är intervjuer och användarenkäter(Wohlin et al., 2012). Men eftersom hastighetsskillnaden mellan databassystem kan vara svår till och med omöjlig för en människa att märka kommer datainsamlingen ske teknikbaserat genom lagring av svarstider som sedan jämförs. Den manipulerade faktorn kommer vara den databashanterare som används vid lagringen av datan men även mängden data som lagras per session, likt experimentet utfört av Damodaran, Salim & Vargese (2016) i vilket de använde sig av en stor mängd data innefattande 100, 1000, 10000 och 25000 rader vid jämförelse av hastigheten mellan MongoDB och MySQL vid inserts och queries. För att utföra detta experiment kommer användningsdata genereras med hjälp av automatiserade användare i form av javascript. Den data som är relevant för att genomföra *User identification* och *Session identification,* som Reddy, Reddy & Sitaramulu(2009) beskriver i steget förbehandling av data, kommer sedan att lagras i de olika databaserna vilket resulterar till att *Data cleaning* steget indirekt även det genomförs. Exekveringstiden för lagringen kommer sedan reflekteras i svarstiden för webbsidan vilket är den variabel mätningarna utförs på.

Fallstudie vore en alternativ undersökningsmetod för denna studie, vilket innebär att vi utför undersökningen i sin verkliga kontext(Wohlin, C et al. 2012), dvs att man testar de hur de olika databassystemen påverkar svarstiden vid lagring av användningsdata på en redan existerande webbsida med riktiga användare. Vilket inte är praktiskt i detta fall då det försvårar egenskapen att isolera de förändrade faktorerna, eftersom vi då även måste ta hänsyn till användarnas varierande beteénde samt uppkopplingshastigheter

En alternativ faktor att manipulera är antalet användare som simuleras att använda sidan synkront, dvs. att flera användare försöker kommunicera med databassystemet parallellt. Vilket potentiellt kan bidra till helt annorlunda resultat än detta experiment då dokumentbaserad datalagring inte prioriterar prestanda vid synkron läs och skrivning (JMTauro, Aravindh & Shreeharsha 2012).

4.2 Etiska aspekter

Testanvändare i experimentet kommer att vara automatiserade med hjälp av javascript vilket innebär att den data som kommer att lagras inte påverkas av bland annat GDPR. På grund av de automatiserade användarna behöver vi inte heller ta hänsyn till de fyra nyckel principerna Wohlin et al. (2012) diskuterar angående etik i experiment; Att användare måste ge informerat samtycke. Att studien måste ha vetenskapligt värde som motiverar användarna att utsätta sig för riskerna att delta i studien. Att sekretess måste upprätthållas vad gäller känslig information. Och att utvärdera risker och fördelar för testanvändarna, varav fördelarna måste uppväga riskerna.

En potentiell nackdel med automatiserade användare kan vara att det inte exakt simulerar verklig användning av systemet och därav kan påverka mätresultaten, om exempelvis systemet anpassar sig till upprepad användning.
De program som kommer att användas i experimentet är alla open source; MySQL Workbench, MongoDBCompass, XAMPP, Visual Studio Code och Chrome-tillägget Tampermonkey.

För att utesluta externa faktorer så som uppkopplingshastighet kommer experimentet utföras i en lokal testmiljö i en lokal webbserver. En stor mängd mätpunkter kommer att användas för att styrka trovärdigheten i experimentet. Återupprepbarhet av experimentet möjliggörs eftersom koden till artefakten kommer att laddas upp till github, inkluderande bland annat javascript-,php- och html filer. På grund av användandet av Open Source program i kombination med att filerna laddas upp till github förenklas återupprepbarheten av experimentet.

5. Genomförande

5.1 Litteraturstudie

Utöver den litteratur och de studier som nämnts tidigare i rapporten kompletterades de med ytterligare litteratur för att skapa en uppfattning om hur artifakten skulle byggas samt stod som underlag för beslut tagna under utvecklingen.

Boken *Php web services, 2nd edition*(2016) skriven av Lorna Jane Mitchell och publicerad av O'Reilly Media, inc. beskriver bland annat utvecklingen av web service med hjälp av php, samt när och hur man bör använda sig av json data. I boken beskrivs även hur filbaserade requests sker med hjälp av bland annat file get content.

Microservices.io är en webbapplikation skapad av Chris Richardson vars syfte är att hjälpa klienter världen över att migrera till microservice arkitektur, på sidan kan man bland annat läsa om för och nackdelar av att använda sig av en samlad databas för flera mikrotjänster. Utöver det beskrivs även tillvägagångssätt för att uppnå microservices samt beskrivningar av vad de system innebär.

På php.net kan man hitta manualer för php samt dess olika drivers och extensions, vilket kan användas för bland annat kommandon för den senaste php mongo drivern, då majoriteten av guider fortfarande använder kommandon för en utfasad driver.

För lagringen av användningsdata samt olika metoder att lösa problem så som att skapa en klass för uppkoppling mot databasen användes stackoverflow.com, en sida för frågor och svar för programmerings entusiaster samt för professionella programmerare. Stackoverflow.com användes även för att utforska olika angreppssätt vid eventuella problem och buggar som dykt upp under utvecklingen av applikationen men även vid utvecklingen av scriptet som simulerar användare för testerna.

Boken *Experimentation in Software Engineering*(2012) skriven av Wohlin, C., Runeson, P., Höst, Ohlsson, Regnell & Wesslén beskriver hur man genomför experiment, olika faktorer som bör tas hänsyn till samt de etiska aspekter som ett experiment medför.

5.2 Progression

För att utföra experimentet i en lokal miljö installerades Xampp för att köra en lokal webbserver, med Xampp tillkommer även extensions för att använda MySQL som lagringsstrategi för denna lokala webbserver. För att installera MongoDB som alternativ databashanterare hämtades den senast släppta versionen av 'Mongo driver' till windows från pecl.php.net. Mongo drivern kopierades sedan in i xamps php extension bibliotek och registrerades i php.ini filen. Tillägget av extension visas i figur 12.

```
extension=php_MongoDB.dll
```

Figur 12: Registrering av MongoDB.dll i php.ini

Sedan installerades MongoDB för windows samt MongoDB compass som verktyg för att hantera databasen.

Innan applikationen samt dess struktur började utformas togs först beslutet att det mikrotjänsterna skulle hantera var hämtning, sökning, bokning samt avbokning av en arbiträr resurs. Resursen i fråga beslutades vara former av olika slag, som sedan lagrades i MySQL databasen. Det lagrades former av olika typer så som 'Square', 'Triangle' och 'Circle' i storlekarna 'Small','Medium' och 'Large'. Utöver storlek och typ lagrades även färg men även Id för resurserna då de behövde ett unikt värde vid exempelvis bokning. Lagringen av dessa former i MySQL dabatasen genomfördes med hjälp av MySQL editorn 'MySQL Workbench'. Tabellen utformades enligt figur 13.

```
create table forms (
    idForms int,
    typeForms varchar(255),
    colorForms varchar(255),
    sizeForms varchar(255),
    PRIMARY KEY(idForms)
);
```

Figur 13: Tabell för resursen 'Forms' i MySQL.

Då bokningar skulle genomföras skapades även en tabell för bokningarna där formens unika ID bokades samt ett ständigt ökande värde för boknings id. Se figur 14.

```
create table bokningar(
    idBokning int AUTO_INCREMENT,
    idForms int,
    foreign key(idForms) references forms(idForms),
    primary key(idBokning)
);
```

Figur 14: Tabell för bokningar av resursen 'Forms' i MySQL.

Tabellen med Former fylldes sedan med diverse former varav 9st av typen 'square' i olika storlekar och färger, 9st av typen 'triangle' och 9st av typen 'circle'. Se figur 15.

```
insert into forms(idForms,typeForms,colorForms,sizeForms)
Value(111,'square','blue','big');
insert into forms(idForms,typeForms,colorForms,sizeForms)
Value(112,'square','blue','medium');
insert into forms(idForms,typeForms,colorForms,sizeForms)
Value(113,'square','blue','small');
```

Figur 15: inserts i tabellen 'forms'.

För att skapa en uppfattning om hur applikationen skulle struktureras skapades 'Applikation.php' filen innehållande olika html form element vars uppgift var att tillkalla de eventuella mikrotjänsterna. Det skapades Form element för att hämta alla former, för att hämta alla bokningar, för att söka efter specifika former samt för att söka efter specifika bokningar. Vilket syns i figur 16.

```
    <h1> Hämta former</h1>
       <form action='form-service.php' method='get'>
        <input type='submit' value='hämta alla former'>
    </form>
```

```
    <h1> Hämta Bokningar</h1>
    <form action='Bokning-service.php' method='get'>
        <input type='submit' value='hämta alla bokningar'>
    </form>
    <h1> Sök efter Form</h1>
    <form action='formsearch-service.php' method='get'>
        <input type='search' Placeholder='sök efter form' value=''>
        <input type='submit' value='sök'>
    </form>
    <h1> Sök efter Bokning</h1>
    <form action='bokningsearch-service.php' method='get'>
        <input type='search' Placeholder='sök efter bokning' value=''>
        <input type='submit' value='sök'>
    </form>
```

Figur 16: Form element för att hämta de olika .php filerna

Till en början skickade dessa form-element användaren till php filer som utförde dessa handlingar. Dvs filen 'form-service.php' utförde en query mot databasen och skrev sedan ut resultaten. Innan det var möjligt dock var en uppkoppling mot databasen tvungen att skapas, vilket utfördes genom en php fil för databasen innehållande en klass för databas med funktionen getConnection, för att undvika att behöva återupprepa attribut så som hostname, databasens namn etc. databas.php filen inkluderades i de filer som krävde en uppkoppling och funktionen getConnection användes istället. Se figur 17.

```
<?php
class Database{

    // Database credentials
    private $host = "localhost:3307";
    private $db_name = "examensarbete2021";
    private $username = "root";
    private $password = "";
    public $conn;

    // Database connection
    public function getConnection(){
        $this->conn = null;

        try{
                    $this->conn = new PDO("mysql:host=" . $this->host . ";dbname=" .
$this->db_name, $this->username, $this->password);
            $this->conn->exec("set names utf8");
        }catch(PDOException $exception){
            echo "Connection error: " . $exception->getMessage();
        }

        return $this->conn;
    }
}
```

Figur 17: Databas.php, fil med funktion för att skapa uppkoppling till databasen

Med en uppkoppling till databasen var php filerna för att hämta former, hämta bokningar och sökning av dessa redo att utformas. Hämtningen av former och bokningar utfördes

genom att filen skickade en sql query till databasen och itererade ut resultaten i tabeller. Se figur 18 och 19.

```
include_once 'database.php';

$database = new Database();
$conn = $database->getConnection();
$sql = "SELECT * FROM forms";
$result = $conn->query($sql);
$row = $result ->fetchall(PDO::FETCH_ASSOC);
```

Figur 18: Form-service.php databas connection och query.

```
    <div id='formdiv'>
        <table>
            <th> olika former </th>
            <?php
                foreach ($row as $data){
            ?>
            <tr>
                <td><?= $data['idForms']?></td>
                <td><?= $data['typeForms']?></td>
                <td><?= $data['colorForms']?></td>
                <td><?= $data['sizeForms']?></td>
            </tr>
            <?php
            }
            ?>
        </table>
```

Figur 19: Form-service.php iterering och utskrivning av resultat

För att implementera sökningsfunktionen användes HTTP-verben POST och GET där tjänsten för sökning istället för att hämta alla former och bokningar utförde sql queries som hämtade formerna och bokningarna som stämde överens med variablen som skickades med i POST verbet. Se figur 20 & 21.

```
$sql = "SELECT * FROM forms where '".$_GET['user_search']."' in (idForms, typeForms,
colorForms, sizeForms) ";
$result = $conn->query($sql);
$row = $result ->fetchall(PDO::FETCH_ASSOC);
```

Figur 20: Formsearch-service.php query mot databasen.

```
$sql = "SELECT * FROM bokningar where '".$_GET['user_boksearch']."' in (idForms) ";
$result = $conn->query($sql);
$row = $result ->fetchall(PDO::FETCH_ASSOC);
```

Figur 21: Bokningsearch-service.php query mot databasen.

Likt Form-service och bokadeForms-serivce itererades resultaten av sökningarna ut. Skillnaden mellan itereringen av resultat var att sökresultaten även itererade ut en form i varje tabell-rad med den underliggande tanken att kalla på ytterligare tjänster för bokning samt avbokning av form-resurserna. Denna implementerades även i resultaten vid hämtning av alla former och bokningar. Se figur 22.

```
<form action='Avbokning-service.php' method=post>
<td><input type='text' name='avbokningsobj' value=<?=$data['idBokning']?> readonly ></td>
<td><?= $data['idForms']?></td>
<td><input type='submit' value='avboka'></td>
</form>
```

Figur 22: Bokningsearch-service.php resultat-form för att kalla på avbokning.

Avbokning-service och bokning-service filerna utformades likt bokning-search och form-search filerna med undantaget att de istället för att hämta sökresultat från databasen utförde inserts och delete queries i bokningar-tabellen. Eftersom de inte utförde några sökningar i databasen som resulterade i sökresultat itererade de inte heller ut några tabeller utan utförde endast modiferingen av tabellen med hjälp av Post/get variabeln från form elementet i bokningsearch-service och formsearch-service. Se figur 23 & 24.

```
<?php
include_once 'database.php';
//Create database connection
$database = new Database();
$conn = $database->getConnection();
$query = $conn->prepare("INSERT INTO bokningar(idForms) VALUES(:idForms)");
            $query->bindParam(':idForms',$_GET['bokningsobj']);
            if($query->execute()) {
                echo json_encode('bokning OK');
            }
?>
```

Figur 23: Bokning-service.php connection och insert.

```
$query = $conn->prepare("DELETE FROM bokningar WHERE idBokning = :idBokning");
            $query->bindParam(':idBokning',$_GET['avbokningsobj']);
```

Figur 24: Avbokning-service.php sql query.

Nästa steg i utvecklingen var att med hjälp av dessa tjänster utveckla en applikation som använde sig av webservices för att utföra dessa handlingar och skicka resultaten via json som beskrivs i boken *Php web services, 2nd edition*(2016) och med hjälp av det skriva ut tabellerna med resultat i applikation filen istället för på en anna url.
Det första steget som utfördes var att se till att de olika tjänsterna hämtade json-data istället för att iterera ut tabeller med resultaten. Detta utfördes genom att ta bort all html kod från de olika tjänsterna och istället använda sig av 'json_encode()' funktionen. Se figur 25.

```
<?php
header('Content-Type: application/json');
include_once 'database.php';
$database = new Database();
$conn = $database->getConnection();
$sql = "SELECT * FROM forms";
$result = $conn->query($sql);
$row = $result ->fetchall(PDO::FETCH_ASSOC);

echo json_encode($row);
?>
```

Figur 25: Uppdaterad version av form-service.php som hämtar resultaten till json format

För att göra en GET request till denna webservice initierades variabeln $url med värdet av webservicens url sedan användes php file_get_contents funktionen för att utföra get requesten till den specifika tjänsten och tillslut dekodades json datan tjänsten returnade med hjälp av json_decode funktionen. Se figur 26.

```
$url="http://localhost/form-service.php";
$jsontext = file_get_contents($url);
$rows = json_decode($jsontext);
```

Figur 26: Uppdaterad version av form-service.php som hämtar resultaten till json format

För att använda detta skapades applikationtest.php filen med samma form-element som applikation.php. Skillnaden mellan form-elementen var dock att istället för att form action attributet skickar användaren till de olika php filerna form-service.php, bokadeForms-service.php etc hämtar applikationtest.php.
Resultaten som avkodas och lagras i variabeln rows ekas sedan ut för att skapa de tabeller som tidigare fanns i de olika tjänsterna för form, bokningar och sökresultat. Se figur 27.

```
foreach($rows as $row){
echo "<tr>
      <form action='applikationtest.php' method=post>
      <td><input type='text' name='bokningsobj' value='$row->idForms' readonly ></td>
      <td>$row->typeForms</td>
      <td>$row->sizeForms</td>
      <td>$row->colorForms</td>
      <td><input type='submit' value='boka'></td>
      </form>
    </tr>";
}
```

Figur 27: Applikationtest.php Utskrift av resultaten från form-service.php get request

För att styra vilken webservice som kallades användes isset php funktionen som kontrollerade vilken handling användaren utfört, dvs. att i de fall användaren vill hämta alla former är $_POST[‘visaFormer’] satt och då utförs en get request till form-service.php och sedan avkodas svaret och itererar ut en tabell med alla former. Se figur 28.

```
if (isset($_POST['visaFormer'])){
    $url="http://localhost/form-service.php";
    $jsontext = file_get_contents($url);
    $rows = json_decode($jsontext);
```

Figur 28: get request till form-service då visaFormer är satt av användaren.

För att utföra loggning av användningsdata skapades ytterligare en mikrotjänst; loggning.php och en tabell i databasen; logtable. Tabellen lagrade ursprungligen ett logID som var ett automatiskt ökande värde likt id för bokningar i bokningar tabellen. Utöver det lagrades även användarens sessions ID samt url för den webservice användaren tillkallat vid knapptryck. Se figur 29.

```
create table logtable(
    logID int auto_increment,
```

```
    sessionID varchar(255),
    url varchar(255),
    primary key(logID)
);
```

Figur 29: MySQL tabell för användar-data.

För att hämta session id användes php funktionen session_start och session_id, session_id tilldelades sedan till variabeln SID som sedan skickades till mikrotjänsten för lagring tillsammans med variabeln url för att lagra den mikrotjänst som användaren tillkallat. Se figur 30.

```
    $url2="http://localhost/loggning.php?url=".$url. "&SID=".$SID;
    $jsontext2 =file_get_contents($url2);
```

Figur 30: Get request för loggning av url och session id.

Mikrotjänsten för lagring fungerade likt den tjänst som lagrar bokningar, den använder sig av http requests för att hämta variablerna som sedan lagras i databasen med hjälp av en query för inserts. Se figur 31.

```
<?php
include_once 'database.php';
$database = new Database();
$conn = $database->getConnection();
$query = $conn->prepare("INSERT INTO logtable(sessionID,url) VALUES(:sessionID,:url)");

        $query->bindParam(':sessionID',$_GET['SID']);
        $query->bindParam(':url',$_GET['url']);

            if($query->execute()) {
                echo json_encode('log ok');
            }
?>
```

Figur 31: Loggning.php webbservice för loggning.

Lagringen av användningsdata utökades senare till att även lagra timestamp vid varje användarhandling men även genom ytterligare en tabell som lagrar användarens session med hjälp av ip adress för att identifiera användaren, sessions id för att koppla den data lagrad i logtable till en användare, och timestamps för sessionstart och avslutat session. Se figur 32.

```
create table logtable(
    logID int auto_increment,
    tid timestamp default current_timestamp,
    sessionID varchar(255),
    url varchar(255),
    primary key(logID)
);
```

Figur 32: Uppdaterad MySQL tabell för användar-data.

```
create table user_session(
```

```
    ip varchar(255),
    sessionID varchar(255),
    sessionactive int default 1,
    sessionstart timestamp default current_timestamp,
    sessionstop timestamp default current_timestamp on update current_timestamp,
    primary key(ip, sessionID)
);
```

Figur 33: MySQL tabell för att lagra användar-sessioner

Hämtningen av IP adress utfördes med hjälp av php funktionen $_SERVER[‘Remote addr’] och lagrades tillsammans med session id. Lagring av användarsession utfördes med hjälp av samma mikrotjänst som lagringen av användar- och användningsdata. För att urskilja en användares handling från starten av en användarsession implementerades en if sats i loggningstjänsten som lagrade till user_session tabellen ifall ip variabeln var skickat i http requesten. Se figur 34.

```
if (isset($_GET['ip'])){
$query       =       $conn->prepare("INSERT       INTO       user_session(ip,sessionID)
VALUES(:ip,:sessionID)");

        $query->bindParam(':sessionID',$_GET['SID']);
        $query->bindParam(':ip',$_GET['ip']);

        if($query->execute()) {
            echo json_encode('log ok');
        }
    }
```

Figur 34: Loggning.php för att starta användarsessioner.

För att starta en session skapades en knapp användaren trycker på. I samband med att den klickas kallas loggningstjänsten. En liknande knapp skapades för att avsluta sessioner som utöver ip och SID skickade med en ‘stop’ variabel. Genom en liknande kontroll som kollar om ip är satt kontrolleras även om ‘stop’ är satt, och i de fall då ‘stop’ är satt uppdateras den logg med samma ip adress och sessions id. Vilket medför att kolumnen sessionstop updateras till ett nytt värde och sessionen är avslutad. Se figur 35.

```
if(isset($_GET['stop'])){
$query   =   $conn->prepare("UPDATE   user_session   set   sessionactive   =   '0'   where
sessionID=:sessionID and ip=:ip");

        $query->bindParam(':sessionID',$_GET['SID']);
        $query->bindParam(':ip',$_GET['ip']);

        if($query->execute()) {
            echo json_encode('log ok');
        }
```

Figur 35: Loggning.php för att avsluta användarsessioner.

Eftersom utvecklingen och testningen sker på en lokal dator i en och samma browser kommer Ip adressen att förbli statisk. På grund av detta skapades ett nytt sessionsid i

samband med att sessionstop knappen klickats genom php funktionen session_regenerate_Id.

Nästa steg i utvecklingen var att skapa versioner av de olika mikrotjänsterna som använde sig av MongoDB istället för MySQL. För att göra detta skapades först en databas med hjälp av MongoDB Compass, i databasen skapades sedan 4st collections; Forms, bokningar, loggar & user_session. Eftersom MongoDB är flexibelt i lagringen skapades inget schema för att kontrollera formatet på den lagrade datan. Istället för att kontrollera vilken kod som skulle köras i varje mikrotjänst beroende på om det var MongoDB eller MySQL som var det aktiva databassystemet skapades helt nya mikrotjänster för MongoDB. Mongo.php filer för hämtning och sökning av former och bokningar samt filer för bokning och avbokning skapades. Eftersom kopplingen till MongoDB databasen sker genom en rad kod togs beslutet att inte skapa en mongo-database.php fil. För att hämta alla former med mongo-form-service användes MongoDB\Driver\Query php kommandot som sedan exekverades med hjälp av executeQuery funktionen. Eftersom MongoDB returnerar data i Bson format itererades först svaren till array innan json_encode funktionen användes. Se figur 36.

```
<?php
header('Content-Type: application/json');

    $mng = new MongoDB\Driver\Manager("MongoDB://localhost:27017");
    $qry = new MongoDB\Driver\Query([]);

    $rows = $mng->executeQuery("ExamensarbeteMongoDB.Forms", $qry);

    echo json_encode(iterator_to_array($rows));

?>
```

Figur 36: Mongo-form-service.php

Den största skillnaden mellan MongoDB versionerna av mikrotjänsterna utöver syntax skillnaden var att det inte går använda sig av auto increment på lagrade värden utan att skapa en javascript funktion för det. Vilket innebär att de fält som lagrar log-id och boknings-id i MySQL blev tvungna att bytas ut. Eftersom boknings id används som identifierare för bokningarna samt som parameter vid avbokning användes istället det automatiserade _id{$oid()} fältet, då varje lagrat dokument i MongoDB automatiskt får ett unikt objectid. Se figur 37.

```
<?php
header('Content-Type: application/json');
    $q=$_GET['avbokningsobj'];
    $bulk = new MongoDB\Driver\BulkWrite;
    $doc = [
        "_id"=> new MongoDB\BSON\ObjectID($q),
    ];
    $bulk ->delete($doc);
    $mng = new MongoDB\Driver\Manager("MongoDB://localhost:27017");
    $result = $mng->executeBulkWrite("ExamensarbeteMongoDB.bokningar", $bulk);
?>
```

Figur 37: Mongo-avbokning-service.php med _Id som identifierare

Eftersom _Id består av en Std klass skapades en array som först encodar den till json format och sedan decodar den för att hämta object id variabeln i klassen vid ut-ekning av tabellerna som innehåller bokning och avbokningsknapparna. Se figur 38.

```
$array = json_decode(json_encode($row->_id),true);
echo '<td><input type="text" name="avbokningsobj" value="'.$array['$oid'].'" readonly
></td>';
```

Figur 38: avkodning av std klass för att komma åt objekt id värdet vid ekning av tabeller.

För att applikationen ska veta vilka mikrotjänster som ska användas skapades variabeln 'mongo' som tilldelades antingen värdet 1 eller 0, då variabelns värde var 1 användes MongoDB mikrotjänsterna och då variabelns värde var 0 användes MySQL mikrotjänsterna. Detta kontrollerades genom diverse if-satser som körde kod beroende på mongo variabelns värde. Se figur 39.

```
if($mongo==0){

$url="http://localhost/formsearch-service.php?user_formsearch=".$_POST['user_formsearch']
;
$jsontext = file_get_contents($url);
$rows = json_decode($jsontext);

$url2="http://localhost/loggning.php?url=".$url. "&SID=".$SID;
$jsontext2 =file_get_contents($url2);
}
elseif($mongo==1){

$url="http://localhost/Mongo-formsearch-service.php?user_formsearch=".$_POST['user_formse
arch'];
$jsontext = file_get_contents($url);
$rows = json_decode($jsontext);

$url2="http://localhost/Mongo-loggning.php?url=".$url. "&SID=".$SID;
$jsontext2 =file_get_contents($url2);
}
```

Figur 39: Ifsats som kallar på formsearch och loggning beroende på mongo variabeln.

Inför pilotstudien skapades ett script i browser-tillägget Tampermonkey. Scriptets uppgift var dels simulera en användare genom att utföra handlingar på hemsidan, men även att mäta responstiden från användarhandling till sidladdningen var genomförd. Detta skedde genom att scriptet skapar en variabel för att lagra tiden vid scriptstart, samt att skapa en variabel för att lagra tiden till localstorage innan sidan laddas om. Scriptet jämför sedan mellanskillnaden på dessa tider för att avgöra tiden för sidladdning. Scriptet lagrar denna uträkning i localstorage, för varje exekvering av scriptet. Se figur 40, 41 & 42.

```
var start1=new Date();
start1 = start1.getTime();
localStorage.setItem("Start",start1);
submit[0].click();
```

Figur 40: Lagring av tid innan sid-laddning till localstorage.

```
var stop=new Date();
stop = stop.getTime();
```

Figur 41: Lagring av tid vid ny sidladdning.

```
var start=new Date();
start = start.setTime(localStorage.getItem('Start'));
var delta=stop-start;
var str=localStorage.getItem("theData");
str+=', ' +delta;
```

Figur 42: Uträkning och lagring sidladdning.

För att simulera en användare på sidan skapas en counter variable som sparas till localstorage, denna avgör vilken handling som ska ske på sidan och ökar med 1 för varje genomkörning av scriptet. För att simulera sessioner skapades session variabeln som även den sparades till localstorage. Efter 15 handlingar genomförts och counter variabeln därmed nått 15, ökar session variabeln och counter variabeln återställs till 0. Se figur 43, 44 & 45.

```
var cnt=parseInt(localStorage.getItem("Counter"));
var session=parseInt(localStorage.getItem("sessioncounter"));
```

Figur 43: Counter och session hämtas från localstorage

```
if(cnt==15){
    cnt=0;
    session++;

}else{
    cnt++;
}
localStorage.setItem("sessioncounter",session);
localStorage.setItem("Counter",cnt);
```

Figur 44: Counter och session värde förändras och lagras till localstorage

```
if(cnt==0){
    submit=[];
    submit.push(document.getElementById('sessionstart'));
}
if(cnt==1 || cnt==3 || cnt == 5 || cnt==7){
   submit=[];
   submit.push(document.getElementById('visaFormer'));
}else if(cnt==2 || cnt==4 || cnt==6 || cnt ==8 || cnt==10){
   submit=document.getElementsByClassName('bokningsubmit');
}else if(cnt==9){
   document.getElementById('user_formsearch').value='triangle';
   submit=[];
   submit.push(document.getElementById('Fsearchsubmit'));
}else if(cnt==11){
   document.getElementById('user_boksearch').value='111';
   submit=[];
   submit.push(document.getElementById('Bsearchsubmit'));
}else if(cnt==12 || cnt==14){
   submit=document.getElementsByClassName('avbokningsubmit');
}else if(cnt==13){
```

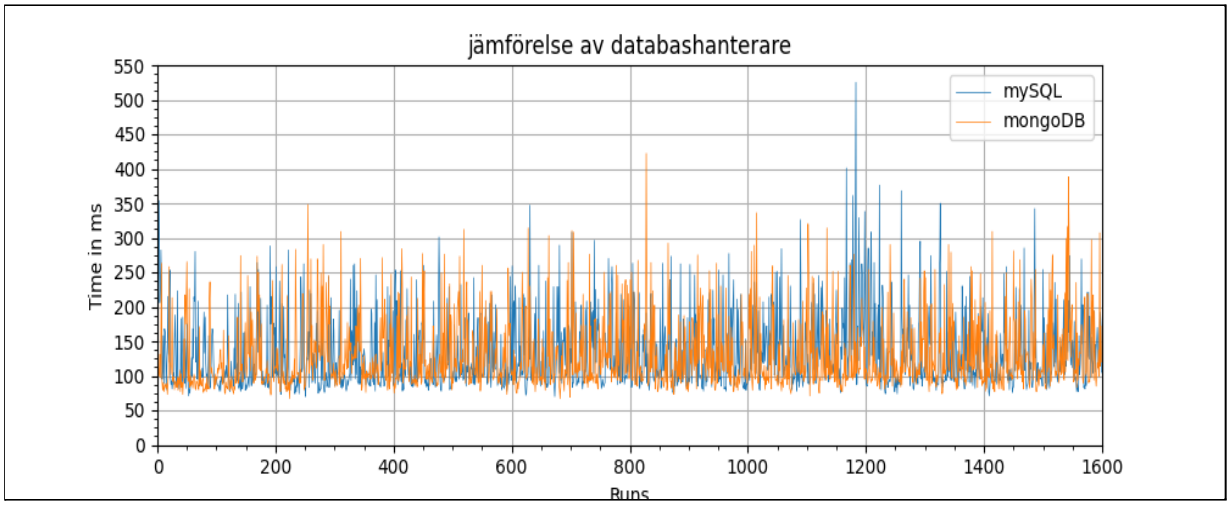
```
   submit=[];
   submit.push(document.getElementById('visaBokningar'));
}else if(cnt==15){
   submit=[];
   submit.push(document.getElementById('sessionstopp'));
}
```

Figur 45: Counter variabeln kontrollerar handlingarna.

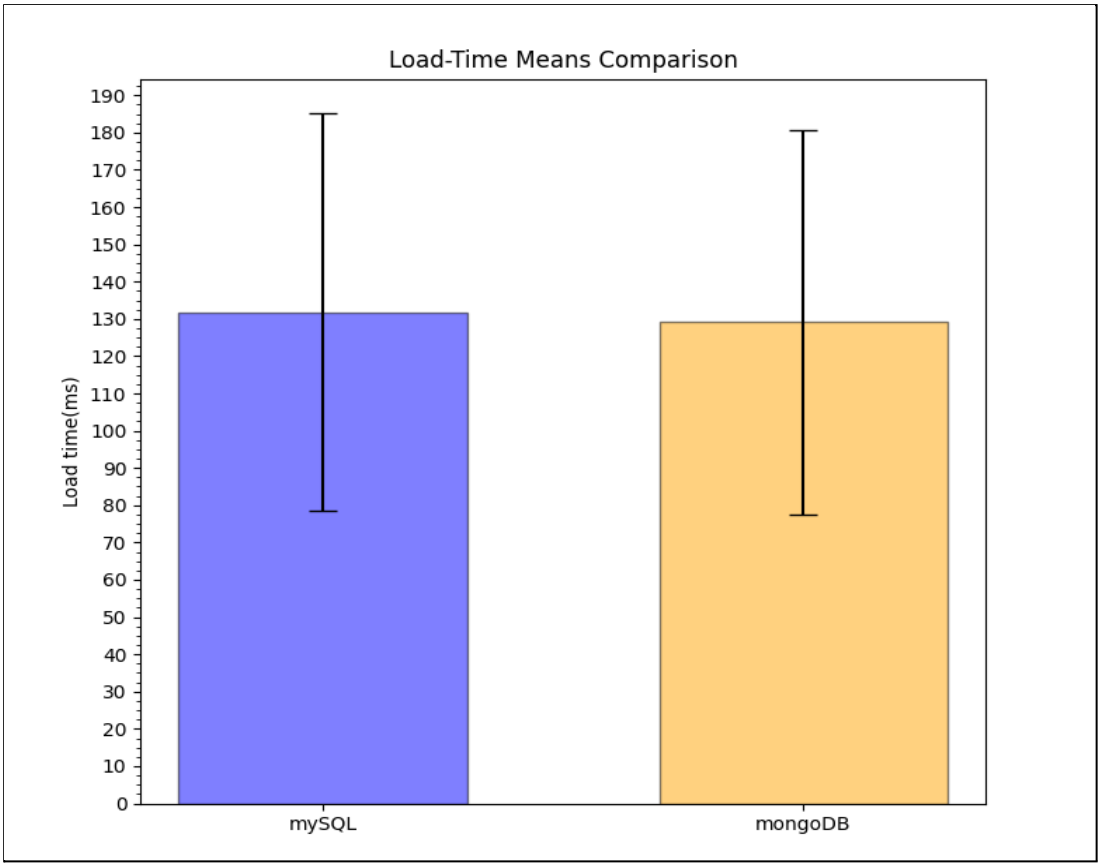
5.3 Pilotstudie

En pilotstudie genomförs vars syfte är att utvärdera den valda metoden och hur väl anpassad den är för hypotesen etablerad i problemformuleringen. Med hjälp av pilotstudien kan man säkerställa att mätningarna som genomförs fungerar korrekt och mäter den tänkta variabeln. I pilotstudien används ett script för att simulera användare på applikationen, vars handlingar lagras i de olika databaserna. Det utförs mätningar på laddningstiden på applikationen för att avgöra hur denna lagring av användande-data påverkar systemets svarstid. Mätningarna i pilotstudien utförs på en initialt tom databas vad gäller användande-data, och består av 100 sessioner per databasimplementation innehållande 14+2 användarhandlingar per session där +2 innebär in och utloggning. Vilket resulterar i 100 lagrade sessioner och 1400 rader användarhandlingar lagrade i MySQL och 1400 dokument i MongoDB. Dessa 100 sessioner av 14+2 handlingar innebär 1599 mätpunkter då mätningen inte mäter svarstiden på sidan efter sista utloggningen.

Mätresultaten sammanställs och presenteras med hjälp av python till linjediagram, samt stapeldiagram med standardavvikelse. Mätdatan som presenteras förekommer i form av millisekunder och presenterar resultaten av 1599 mätpunkter samt medelvärdet av de mätpunkterna. Se figur 46 & 47.



Figur 46: Linjediagram för mätning av 100 användarsessioner



Figur 47: Stapeldiagram för mätning av 100 användarsessioner

Resultaten av pilotstudien påvisar att mätningar av svarstiden på hemsidan är genomförbara trots att det inte går att dra några konkreta slutsatser av mätresultaten då det endast visar marginell skillnad mellan de olika databashanterarna. Vilket eventuellt kan ändras med en större mängd mätpunkter samt med mer lagrad data. Resultaten av pilotstudien visar medelvärde av svarstiden följande: 131.84 millisekunder för MySQL och 129.15 millisekunder för MongoDB. Med 53.29 millisekunder i standardavvikelse för MySQL och 51.57 millisekunder i standardavvikelse för MongoDB.

6. Utvärdering

Resultatet av pilotstudien tydde på att det går utföra experimentet för att mäta den påverkan lagringen av användningsdata har på svarstiden på applikationen. För att genomföra experiementet på en större skala, ökas antalet användarsessioner som simuleras per mätning. En modifikation på mätningarna var dock tvungen att genomföras då varje simulerad användare bokar samt utför handlingar för att hämta alla bokade resurser. På grund av att alla simulerade användare bokar ett X antal resurser ökar antalet bokade resurser, vilket innebär att laddningstiden på sidan exponentiellt ökar ju större mätningar som genomförs. Eftersom applikationen använder sig av MongoDb i ena mätningarna och MySQL i de andra så vet vi inte om resultatet av mätningarna beror på att den ena databasen hämtar bokningarna snabbare än den andra. För att minimera denna påverkan så modiferas scriptet så de simulerade användarna endast bokar resurser och avstår från att hämta alla bokade resurser, då det är påverkan av laddningstiden vid lagring av användningsdata som är målet med mätningarna och inte vilket databassystem som är effektivast vid hämtning av data. Utöver den ändringen införs även en baseline för mätningarna där svarstiden på applikationen mäts då lagring av användningsdata inte genomförs. Det genomförs mätningar mot Applikationen då den använder sig av MongoDB som databassystem vid baseline testet samt mätningar då applikationen använder sig av MySQL vid baseline testning för att mäta påverkan av lagring av användningsdata har på sidladdningen.

6.1 Presentation av undersökning

För att vidareutveckla mätningarna utförda i pilotstudien på en större skala utfördes mätningar som bestod av 250, 500 och 1000 simulerade användarsessioner. Mätningarna utfördes initialt mot en tom databas, utan bokningar, utan lagrade användarsessioner och lagrade användarhandlingar. Dessa värden ökade i och med att mätningarna genomfördes och tömdes då byte av databassystem genomfördes. Dvs. då mätningarna mot applikationen utan lagring av användande data genomfördes startades experimentet med en tom databas vad gäller bokningar. Efter tre mätningar-serier genomförts mot appliktionen utan lagring av användande data tömdes databasen åter igen. Därefter utfördes mätningar mot applikationen med lagring av användande data i en MongoDB databas, och till sist på mot applikationen med lagring i en MySQL databas. De olika mätserierna innebär sammanlagt 28000 mätpunkter per applikation, då 250 användarsessioner innefattar 4000 lagrade handlingar, 500 sessioner innefattar 8000 handlingar och 1000 sessioner innefattar 16000 handlingar.
Resultaten av de olika mätserierna sammanställdes och utvärderades sedan med hjälp av linjediagram, stapeldiagram, anova-tester och tuckey-tester.

6.1.1 Hårdvaruspecifikationer

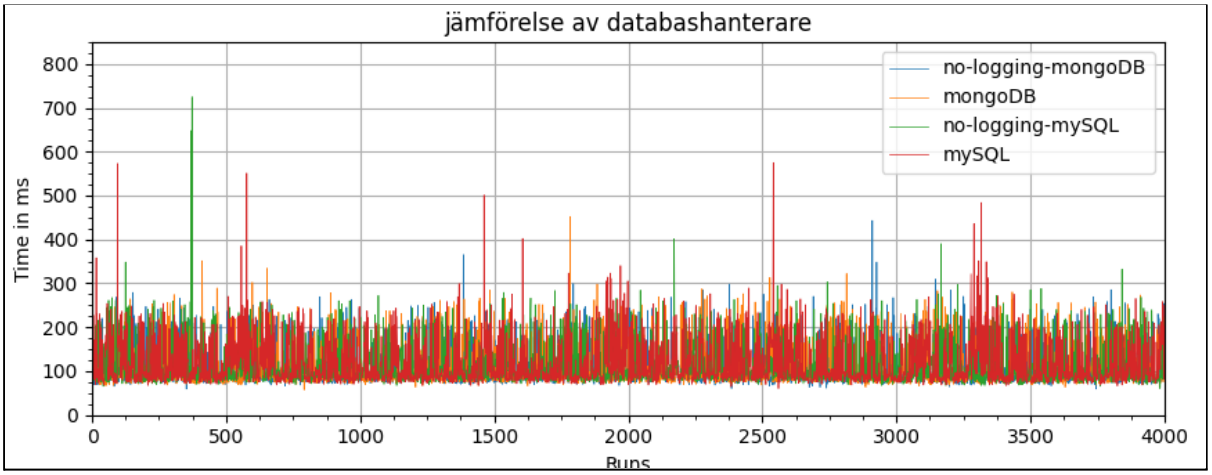
Nedan följer en lista av komponentspecifikationer för hårdvaran använd i experimentet

Processor	Intel(R) Core(™) i5-4690k @ 4.70GHz

Installerat RAM-minne	DDR3 8,00GB
Moderkort	MSI Z97S SLI Krait Edition, Socket-1150

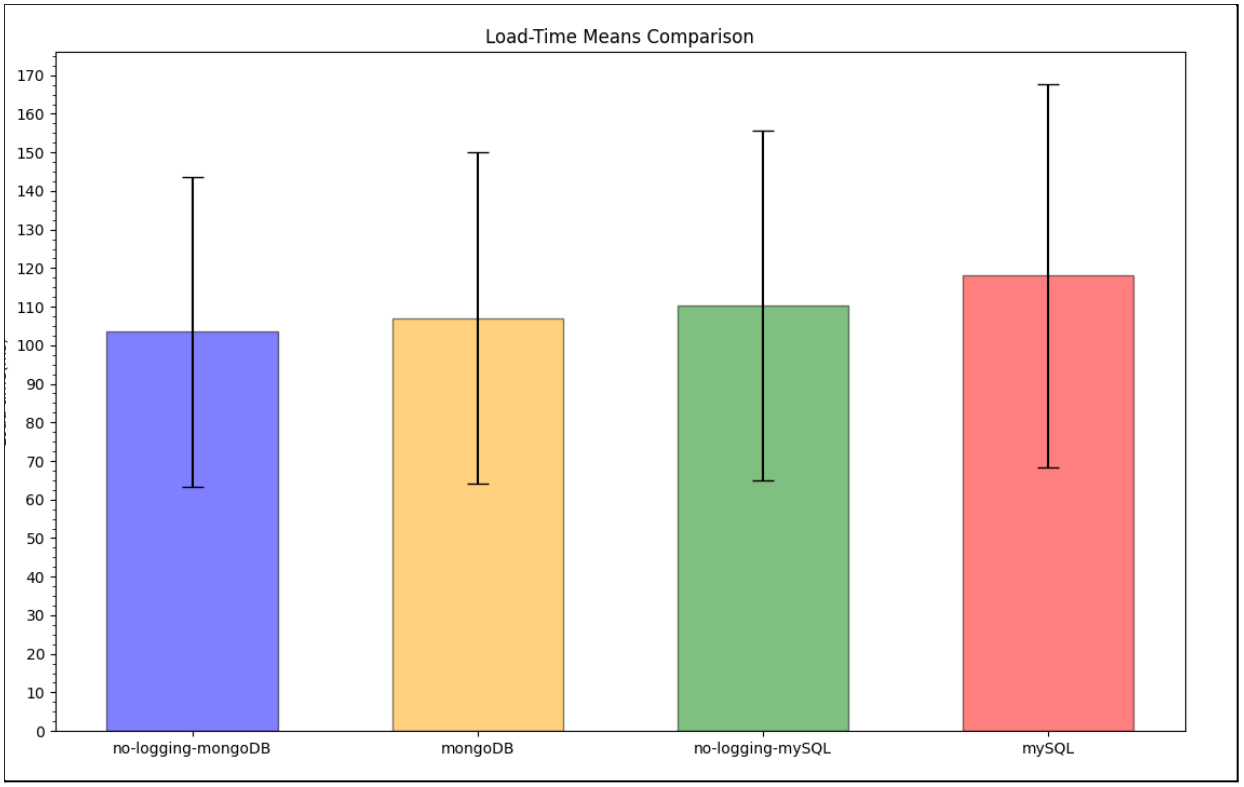
6.1.2 Mätserie 250 sessioner

I figur 48 presenteras mätserie 1, 250 sessioner för respektive variant av applikationen. Varje mätpunkt presenteras med färgade linjer där X-axeln presenterar antalet mätpunkter och Y-axeln tiden för varje mätpunk. Tiden mäts i millisekunder.



Figur 48: Linjediagram 250 användarsessioner, 4000 mätpunkter.

I linjediagrammet presenteras mätpunkterna för applikationen utan lagring av användande data på en MongoDB baserad applikation med linjer markerade i blå färg. Lagring av användningsdata i MongoDB presenteras med linjer markerade i orange färg. mätningar mot en applikation utan lagring av användningsdata mot en applikation baserad på MySQL presenteras med linjer markerade i grön färg och mätningar på applikationen som lagrar användningsdata i MySQL presenteras med röd färg.



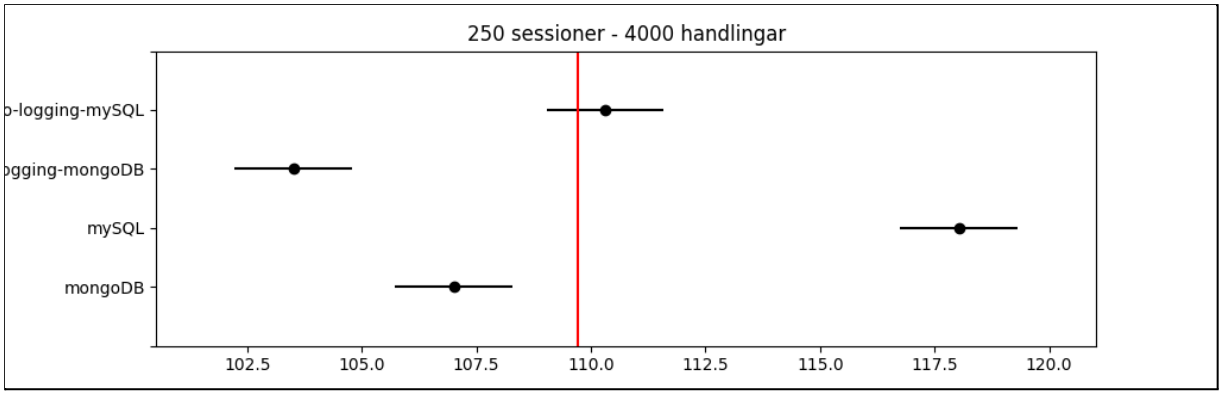
Figur 49: Stapeldiagram 250 användarsessioner, 4000 mätpunkter, med standardavvikelse

I figur 49 presenteras mätserien med hjälp av stapeldiagram inkluderande standardavvikelse. Y-axeln visar tiden i millisekunder. Mätningar av sidladdning mot applikationen utan lagring av användningsdata i en MongoDB baserad applikation presenteras med blå färg, mätningar av sidladdning mot applikationen med lagring av användningsdata mot MongoDB med orange färg och mätningar av sidladdning mot applikationen utan lagring av användningsdata mot en MySQL baserad applikation med grön färg. Mätningar av sidladdningen i MySQL baserad applikation med lagring av användardata presenteras med röd stapel. Varje stapel presenterar medelvärdet av mätpunkterna samt standardavvikelsen. Medelvärdet av mätpunkterna för sidladdning utan lagring av användningsdata i en MongoDB baserad applikation var 103.51 ms med 40.18 ms i standardavvikelse, för lagring mot MongoDB 107 ms med 42.94 ms i standardavvikelse, för lagring mot en MySQL baserad applikation utan lagring av användardata var medelvärdet 110.3 ms och standardavvikelesen 45.24 ms. För lagring mot MySQL 118 ms med 49.65 ms i standardavvikelse. På grund av att resultaten överlappar krävs det ytterligare analys av resultaten genom anova-testning för att avgöra om det är bevisliga resultatskillnader.

ANOVA Statistic 77.08250401449936	p-value 1.688819928482333e-49	The means are different

Figur 50: Anova-test 250 användarsessioner, 4000 mätpunkter.

Anova testet visar att det finns skillnad mellan de olika mätseriernas resultat, för att avgöra vilket resultat som skiljer sig från de andra utförs ett så kallat post-hoc test, tuckey-test.



Figur 51: Tuckey-test 250 användarsessioner, 4000 mätpunkter

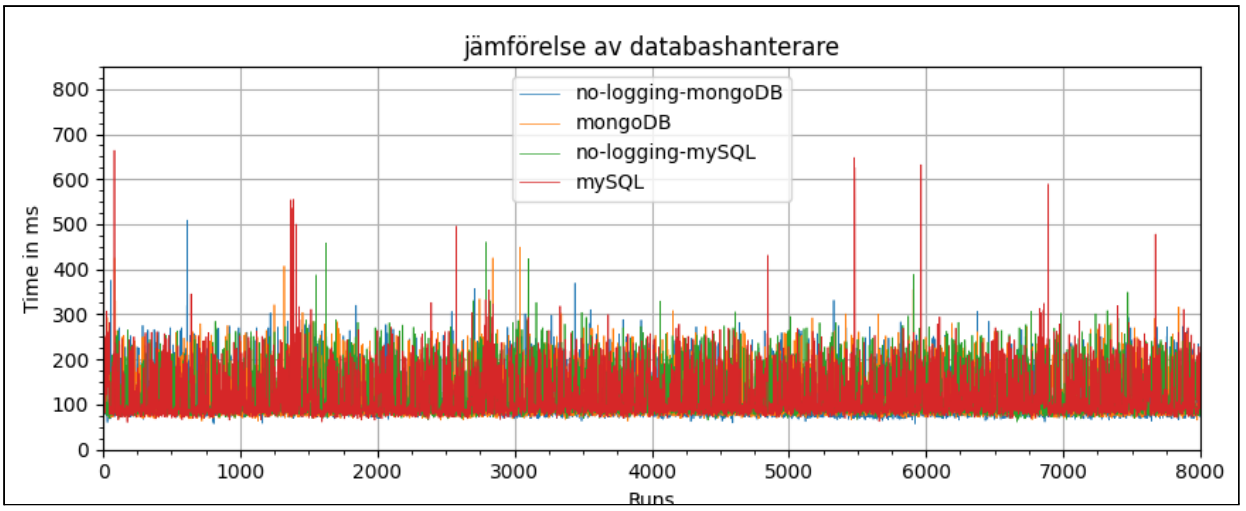
group1	group2	meandiff	p-adj	lower	upper	reject
MongoDB	MySQL	11.0173	0.001	8.4522	13.5823	True
MongoDB	no-logging-MongoDB	-3.4984	0.0026	-6.0634	-0.9333	True
MongoDB	no-logging-MySQL	3.2986	0.0053	0.7335	5.8636	True
MySQL	no-logging-MongoDB	-14.5156	0.001	-17.0807	-11.9506	True
MySQL	no-logging-MySQL	-7.7187	0.001	-10.2838	-5.1536	True
no-logging-MongoDB	no-logging-MySQL	6.7969	0.001	4.2319	9.362	True

Figur 52: Tuckey-test data 250 användarsessioner, 4000 mätpunkter

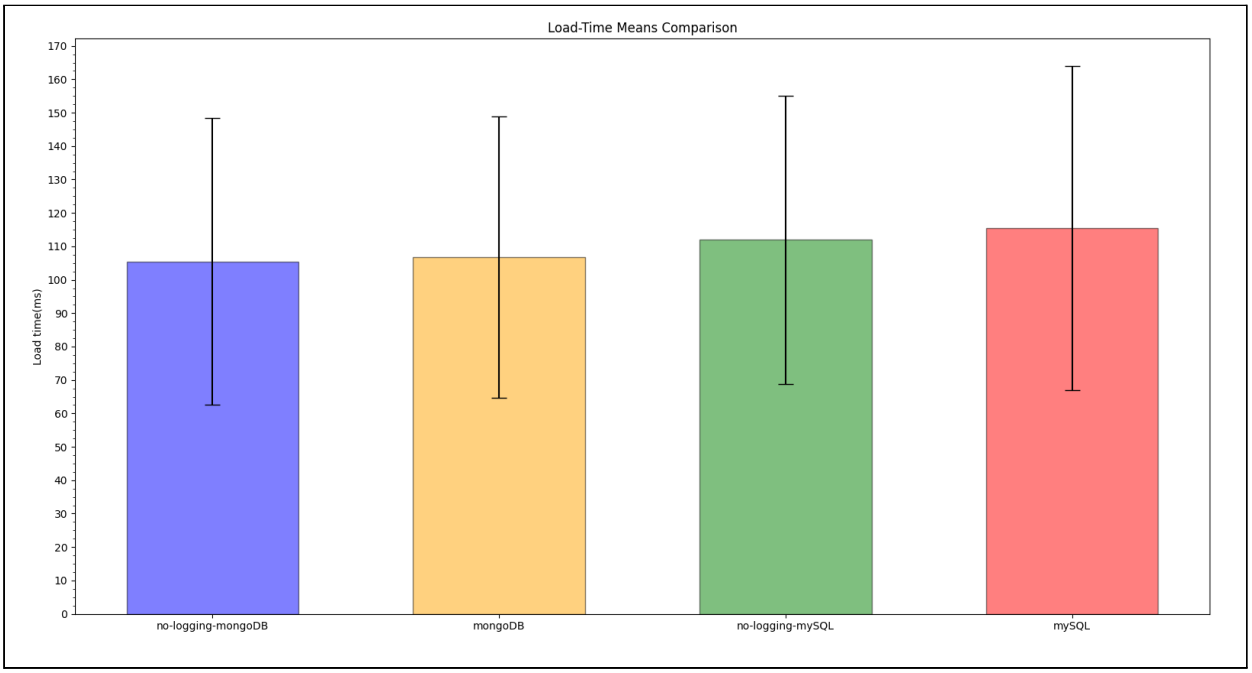
Tuckeytestet används för att avgöra vilken mätserie som skiljer sig från de andra. I figur 51 presenteras mätserien mot applikationen utan lagring av användningsdata mot en MySQL baserad applikation av den översta figuren, den som mäter mot applikationen baserad på MongoDB utan lagring av användningsdata av den näst-översta och den med som mäter mot applikationen med lagring mot MySQL av den näst-understa figuren. Och den som mäter mot applikationen som lagrar användningsdata i MongoDB av den understa figuren.

6.1.3 Mätserie 500 sessioner

Mätserie 2 med 500 sessioner utfördes på samma sätt som mätserien för 250 sessioner. Resultaten presenteras även de med hjälp av samma metoder, linjediagram, stapeldiagram, anova-test och tuckey-test.



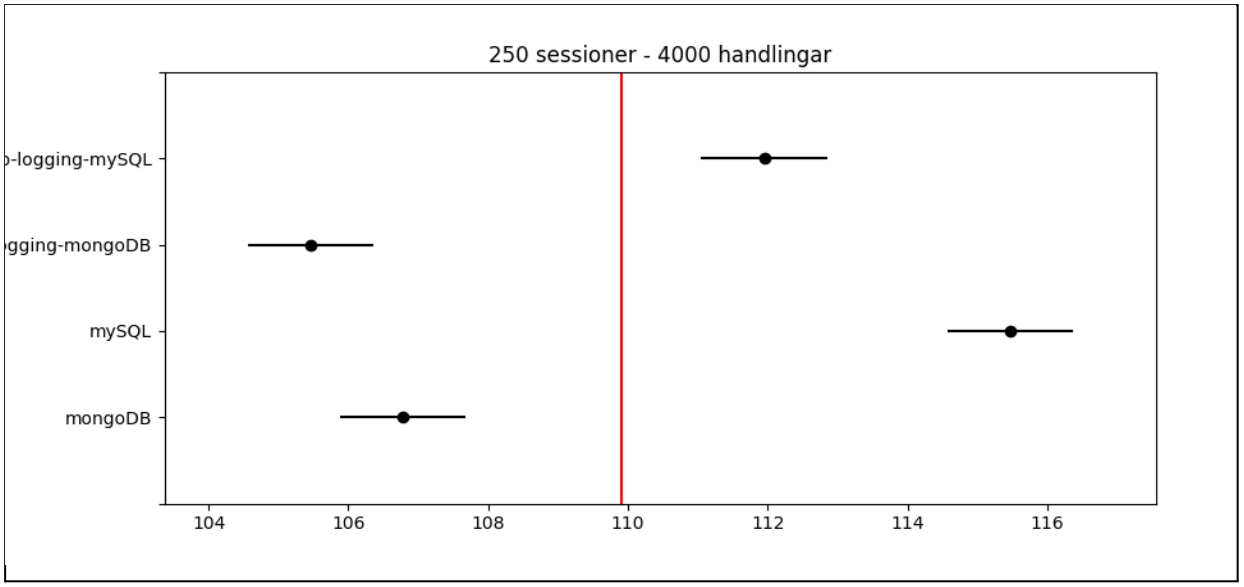
Figur 53: Linjediagram 500 användarsessioner, 8000 mätpunkter.



Figur 54: Stapeldiagram 500 användarsessioner, 8000 mätpunkter.

ANOVA Statistic 88.20715262255733	p-value 7.720251375865455e-57	The means are different

Figur 55: Anova-test 500 användarsessioner, 8000 mätpunkter.



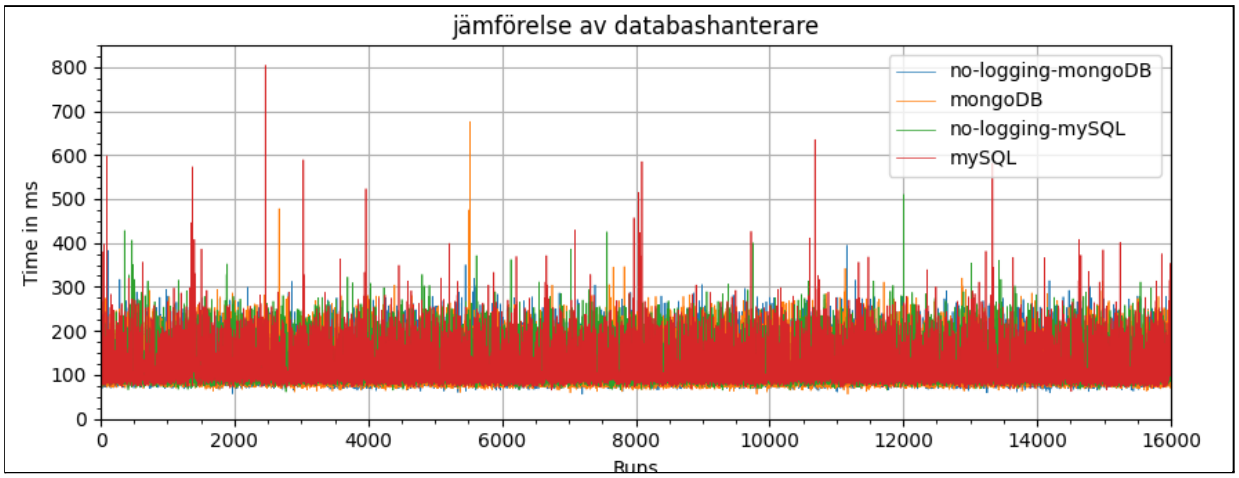
Figur 56: Tuckey-test 500 användarsessioner, 8000 mätpunkter.

group1	group2	meandiff	p-adj	lower	upper	reject
MongoDB	MySQL	8.6958	0.001	6.8985	10.4932	True
MongoDB	no-logging-MongoDB	-1.3222	0.2325	-3.1195	0.4752	False
MongoDB	no-logging-MySQL	5.1699	0.001	3.3725	6.9673	True
MySQL	no-logging-MongoDB	-10.018	0.001	-11.8154	-8.2206	True
MySQL	no-logging-MySQL	-3.5259	0.001	-5.3233	-1.7286	True
no-logging-MongoDB	no-logging-MySQL	6.4921	0.001	4.6947	8.2894	True

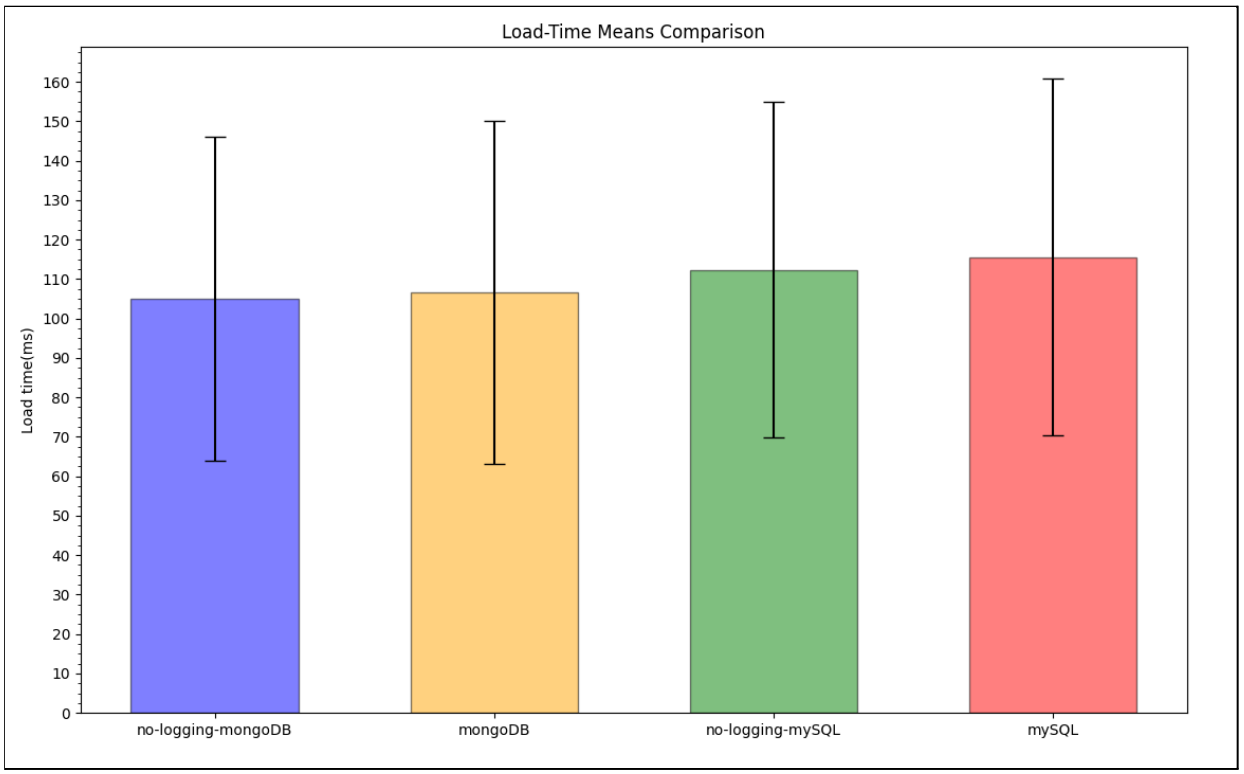
Figur 57: Tuckey-test data 500 användarsessioner, 8000 mätpunkter.

6.1.4 Mätserie 1000 sessioner

Mätserie 3 med 1000 sessioner utfördes på samma sätt som mätserien för 250 sessioner och 500. Resultaten presenteras även de med hjälp av samma metoder, linjediagram, stapeldiagram, anova-test och tuckey-test.



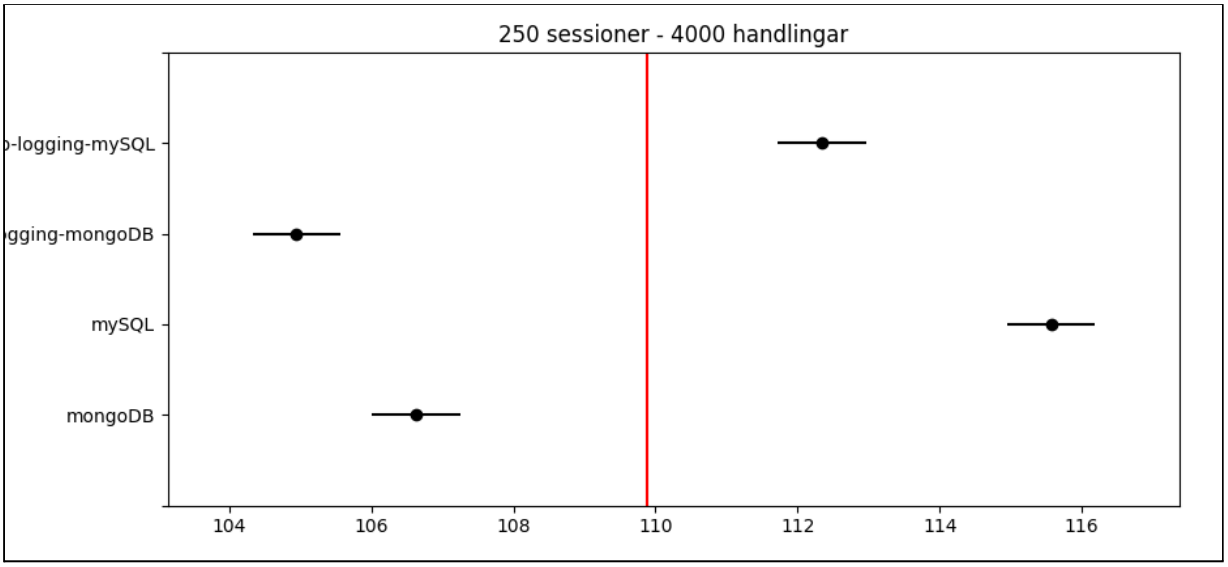
Figur 58: Linjediagram 1000 användarsessioner, 16000 mätpunkter.



Figur 59: Stapeldiagram 1000 användarsessioner, 16000 mätpunkter.

ANOVA Statistic 210.45902672755997	p-value 7.426668695358976e-136	The means are different

Figur 60: Anova-test 1000 användarsessioner, 16000 mätpunkter.



Figur 61: Tuckey-test 1000 användarsessioner, 16000 mätpunkter.

group1	group2	meandiff	p-adj	lower	upper	reject
MongoDB	MySQL	8.9378	0.001	7.6994	10.1762	True
MongoDB	no-logging-MongoDB	-1.6863	0.0026	-2.9247	-0.4479	True
MongoDB	no-logging-MySQL	5.7139	0.001	4.4755	6.9523	True
MySQL	no-logging-MongoDB	-10.6241	0.001	-11.8625	-9.3857	True
MySQL	no-logging-MySQL	-3.2239	0.001	-4.4623	-1.9855	True
no-logging-MongoDB	no-logging-MySQL	7.4002	0.001	6.1618	8.6386	True

Figur 62: Tuckey-test data 1000 användarsessioner, 16000 mätpunkter.

6.2 Analys

Första intrycket av diagrammen som presenterar mätdatan är att det finns viss skillnad på sidladdningen vid lagring av användningsdata. I figur 48 syns det att mätresultaten spikar mycket mer frekvent då MySQL används som lagringssystem. I figur 49 syns det att medelvärdet av svarstiden är något lägre då lagring av användningsdata inte förekommer, då den genomsnittliga laddningstiden för en applikation baserad i MongoDB utan lagring av användningsdata är 103.51 ms, och för lagring mot MongoDB 107 ms. Samma gäller för de

applikationer som använder sig av MySQL som lagringssystem då den genomsnittliga laddningstiden utan lagring av användningsdata var 110.3 ms, och för lagring mot MySQL 118 ms. Vilket tyder på att tiden för sidladdning påverkas mer vid användandet av MySQL som databassystem vid 250 användarsessioner med sammanlagt 4000 lagrade användarhandlingar. Vilket även styrks av tuckey-testet, som påvisar att skillnaden är större vid de fall då MySQL används som lagringssystem.
Vid mätserierna om 500 användarsessioner förekommer även spikade värden i mätdatan mycket mer frekvent vid användandet av MySQL, speciellt då lagring av användningsdata sker, vilket syns i figur 53. Stapeldiagrammet i figur 54 visar medelvärden på 105.45 ms då applikationen baserad i MongoDB används utan lagring av användningsdata. 106.77 ms då lagring av användningsdata sker mot MongoDB applikationen. 111.94 ms för de mätningar mot MySQL baserade applikationen utan lagring av användningsdata och 115.47ms då lagring av användningsdata sker. Till skillnad från mätserien med 250 sessioner visar tuckey-testet i figur 56 och figur 57 att det inte finns någon konkret skillnad mellan de mätningar som genomförts på applikationen som använder sig av MongoDB, medan skillnaden mellan MySQL applikationerna förblir.
I figur 58 visar mätserie 3 med 1000 sessioner även den att spikar förekommer mer frekvent vid användandet av MySQL som databassystem vid lagring av användningsdata. Stapeldiagrammen i figur 59 visar 104.94 ms som medelvärde för tiden vid sidladdningen av MongoDB utan lagring av användningsdata och 106.63 ms då lagring av användningsdata sker. 112.34 ms som medelvärde för MySQL utan lagring av användningsdata och 115.57 då lagring av användningsdata utförs. Tuckey-testet i figur 61 och figur 62 visar att det finns skillnad mellan de resultat utan lagring av användningsdata och då lagring av användningsdata förekommer.

6.3 Slutsatser

Den första slutsatsen som kan antas utifrån analysen av mätningarna är att MongoDB generellt bidrar till snabbare sidladdning specifikt för en applikation lik den som utvecklats för att utföra experimentet i rapporten. Om inte annat i de fall där lagring mot databasen sker då hämtning av data endast skett då den simulerade användaren hämtat de existerande 36 formerna vilket är en väldigt liten volym hämtad data.

Den genomsnittliga sidladdningstiden har varit lägre då MongoDB används som databassystem i alla mätserier, vilket kan vara en produkt av de frekventa spikar som förekommit i de mätningarna som genomförts då MySQL använts som databassystem.

Vad gäller påverkan av sidladdningstid vid lagring av användningsdata ökade medelvärdet av sidladdningen med ~4,22 ms mer vid lagring av användningsdata i MySQL än i MongoDB vid 250 användarsessioner, och vid 500 användarsessioner fanns det ingen mätbar skillnad mellan MongoDB applikationerna medan MySQL applikationens sidladdning genomsnittligen ökade med ~3.52 ms. Vid 1000 användarsessioner ökade även den genomsnittliga tiden vid sidladdning mer för MySQL applikationen än för MongoDB applikationen, i vilken den ökade med ~1.53 ms mer då MySQL användes.

Detta innebär att MongoDB kan vara lämpligare att implementera än MySQL vid användandet av microservice arkitektur i de fall där lagring av användingsdata kommer att förekomma. Utöver det kan även slutsatsen dras att MongoDB kan vara lämpligare att använda även i de fall där lagring av användningsdata inte kommer att ske. Dock är skillnaden marginell då det handlar om enstaka millisekunder, trots de spikar som förekom

vid användningen av MySQL, vilket innebär att en migration till MongoDB från MySQL inte nödvändigtvis är något som behöver prioriteras i redan existerande applikationer.

7. Avslutande Diskussion

7.1 Sammanfattning

Bland webbapplikationer i dagens samhälle trenderar migrationen från användandet av monolitisk arkitektur till microservice arkitektur. Utöver det blir det allt mer vanligt att lagra användares data i form av användningsmönster samt användares personliga information. I en evigt växande World Wide Web genereras en massiv volym användningsdata. Genom att lagra den relevanta data som genereras kan man med hjälp av tillämpning av Web Usage Mining tekniken utveckla system med bättre quality of service (Reddy, Reddy & Sitaramulu, 2013). Menasce(2002) beskriver *Quality of service* som nyckeln att bedöma hur väl webbaserade applikationer uppfyller kundernas förväntningar på två primära åtgärder: tillgänglighet och svarstid. Och att dålig Quality of service i form av lång svarstid leder till frustrerade användare, vilket i sin tur kan leda till förlorade affärsmöjligheter. På grund av vikten svarstiden har för en webbapplikation uppstår frågan hur lagringen av användningsdata påverkar svarstiden i en applikation byggd i den moderna microservice-arkitekturen. Och om den eventuella påverkan lagringen har går att minimera beroende på val av databassystem vid lagring av användningsdatan. För att besvara det beslutades det att testning bör ske vid lagring mot den traditionella relationsdatabasen MySQL och NoSQL databasen MongoDB, vars styrkor bland annat är hastighet vid inmatning och läsning av data.

Frågeställningen som arbetet besvarade var:
påverkar valet av databassystem vid lagring av användningsdata enligt web usage mining tekniken svarstiderna hos en applikation byggd med microservice arkitektur?

Inför experimentet utvecklades en enkel micro-service applikation för bokning av resurser, denna applikation byggdes med en alternativ för lagring via MySQL samt för lagring via MongoDB. För varje handling en användare utförde på applikationen lagrades den relevanta datan enligt Web Usage Mining; User Identification i form av IP-adress, Session identification I form av Session-id och user-clickstream i form av URL och timestamps. Mätningar av svarstiden vid sidladdning genomfördes sedan på 4 olika versioner av applikationen, en byggd med MongoDB som lagringssystem och en med MySQL som lagringsystem, både med och utan lagring av användningsdata. Mätningarna genomfördes med hjälp av ett tekniskt experiment där tiden mättes från användarhandling till färdigladdad sida. Experimentet utfördes genom att simulera användare med javascript i 3 olika mätserier varav mätserie 1 bestod av 250 användarsessioner där varje session bestod av 16 användarhandlingar. Mätserie 2 bestod av 500 av dessa användarsessioner och mätserie 3 bestod av 1000 användarsessioner. Baserat på resultaten av mätdatan presenterade i kapitel 6.1 och analysen av datan i kapitel 6.2 presterade applikationer med lagring mot MongoDB bäst vid lagring av användningsdata, både i hastighet vid sidladdning generellt och i påverkan av tiden vid sidladdning då lagring av användningsdata förekom.

7.2 Diskussion

Utöver frågeställningen angående påverkan som lagringen av användningsdata har på tiden vid sidladdning, visade även experimentet en skillnad vid sidladdning generellt mellan applikationerna beroende på val av databassystem. Applikationen presterade märkbart bättre vid användandet av MongoDB än MySQL i alla tester, oberoende på om lagring av användningsdata förekom. Detta kan bero på de frekventa spikar som förekom under mätningen då MySQL användes och resultaten kan komma att se annorlunda ut i en analys av datan med borttagning av spikar. Eftersom spikarna skedde så frekvent och under alla mätserier togs dock beslutet att spikarna fick förbli.
MongoDB kan vara lämpligare att implementera än MySQL vid utvecklingen av applikationer med microservice arkitektur i de fall där lagring av användingsdata förekommer. På grund av den marginella skillnad vid påverkningen av sidladdningen vid lagring av användningsdata mellan de olika databassystemen föreslår jag att MongoDB endast behöver prioriteras i de fall där man bygger en applikation från grunden, eller i de fall där enstaka millisekunder i längden uppväger resurserna det tar att byta lagringssystem.
I studien som genomfördes av Damodaran, Salim & Vargese (2016) indikerar resultaten att i en applikation för stormarknader, som hanterar ett stort utbud av produkter vilket innebär en omfattande mängd data, var svarstiderna vid inserts marginellt bättre vid användandet av MongoDB jämfört med MySQL.
Skillnaden från experimentet utfört av Damodaran, Salim & Vargese (2016) är bland annat att de mätte svarstiden hos själva databasen och inte sidladdningen hos webbapplikationen, men även att det system de utfört experimentet på baserades på en monolitisk arkitektur. Dock utfördes deras experiment i en redan existerande miljö vilket kan innebära mer reella resultat då applikationen utformad för experimentet i rapporten är en väldigt enkel applikation vars syfte endast var att tillåta bokning av en arbiträr resurs samt att lagra handlingarna användaren utför för att hämta och boka resursen. Utöver simpliciteten hos applikationen kan ytterligare en faktor påverka trovärdigheten i experimentet, vilket är kunskapen jag som utvecklare besitter eller saknar då lagring av användningsdata är helt nytt för mig samt användandet av databashanteraren MongoDB. En utvecklare mer erfaren inom områden kan eventuellt ha optimerat metoden för lagring av användningsdata för att minimera påverkan de olika databassystemen har på sidladdningen vid lagringen. De skulle även potentiellt kunnat använda sig av schemas för lagringen av data i MongoDB, vilket jag inte gjorde. Detta skulle även kunnat påverka resultaten av experimentet. En annan faktor att ha i åtanke är hårdvaran experimentet utförts på, då det utförts i en lokal miljö. De installerade RAM-minnet som används är av den äldre generationen och införskaffades 2014, processorn använd i experimentet är av en äldre generation och även den införskaffades 2014. Vilket innebär att experimentet utförts i en miljö på en hårdvara minst 7 år gammal, vilket potentiellt medför negativ påverkan av svarstiderna.

7.3 Framtida arbete

Experimentet utfört i denna rapport kan användas till grund för vidare undersökningar angående påverkan lagring av användningsdata har på sidladdning i en applikation byggd i en microservice-arkitektur. Ytterligare undersökningar som kan genomföras innefattar bland annat tester mot en redan existerande applikation med mer funktionalitet för att undersöka påverkan i en mer reell miljö. Undersökningar kan även genomföras med hjälp av verkliga användare som alternativ till automatiserade för att undvika upprepning av identiska

användarsessioner. Det skulle även kunna innefatta att fler användare utför testet synkront för att testa påverkan lagringen av användningsdatan har på sidladdningen vid synkron lagring hos de olika databassystemen. Det skulle även kunna utföras undersökning med alternativ metod för lagring, då det i experimentet i arbetet lagrar varje handling separat och kopplar det till en användarsession med hjälp av sessionsid. Ett alternativ vore att lagra handlingarna sessions-vis dvs. att alla handlingar exempelvis lagras som variabler som sedan lagras klumpvis i databasen då en session avslutas. Utöver det kan även undersökningar utföras med fler alternativa databashanterare testas, bland annat andra NoSQL databashanterare så som Cassandra DB, CouchDB och Oracle NoSQL Database. Det vore även intressant att utföra experimentet med hjälp av en modernare hårdvaru-miljö. Ytterligare studier skulle även kunna innefatta tester där systemet använder olika databassystem för backend och för lagringen av användningsdata. Det skulle exempelvis innebära en applikation som använder sig av MongoDB för backend och MySQL för lagringen av användningsdata.

Referenser

Al-Debagy, O. and Martinek, P. (2018) 'A Comparative Review of Microservices and Monolithic Architectures', in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI). 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 000149–000154. doi: 10.1109/CINTI.2018.8928192.

Braun, E. et al. (2017) 'A Generic Microservice Architecture for Environmental Data Management', in Environmental Software Systems. *Computer Science for Environmental Protection. International Symposium on Environmental Software Systems*, Springer, Cham, pp. 383–394. doi: 10.1007/978-3-319-89935-0_32.

Chen, R., Li, S. and Li, Z. (2017) 'From Monolith to Microservices: A Dataflow-Driven Approach', in *2017 24th Asia-Pacific Software Engineering Conference (APSEC). 2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 466–475. doi: 10.1109/APSEC.2017.53.

Damodaran B, D., Salim, S. and Vargese, S. M. (2016) 'Performance Evaluation of MySQL and MongoDB Databases', *International Journal on Cybernetics & Informatics*, 5(2), pp. 387–394. doi: 10.5121/ijci.2016.5241.

Gu, Y. et al. (2015) 'Analysis of data storage mechanism in NoSQL database MongoDB', in *2015 IEEE International Conference on Consumer Electronics - Taiwan. 2015 IEEE International Conference on Consumer Electronics - Taiwan*, pp. 70–71. doi: 10.1109/ICCE-TW.2015.7217036.

Inbarani, H. H. and Thangavel, K. (2006) 'Clickstream Intelligent Clustering using Accelerated Ant Colony Algorithm', in *2006 International Conference on Advanced Computing and Communications. 2006 International Conference on Advanced Computing and Communications*, pp. 129–134. doi: 10.1109/ADCOM.2006.4289869.

JMTauro, C., S, A. and A.B, S. (2012) 'Comparative Study of the New Generation, Agile, Scalable, High Performance NOSQL Databases', *International Journal of Computer Applications*, 48(20), pp. 1–4. doi: 10.5120/7461-0336.

Lawrence, R. (2014) 'Integration and Virtualization of Relational SQL and NoSQL Systems Including MySQL and MongoDB', in *2014 International Conference on Computational Science and Computational Intelligence. 2014 International Conference on Computational Science and Computational Intelligence*, pp. 285–290. doi: 10.1109/CSCI.2014.56.

Meenatchi, Dr. V. T. et al. (2018) 'A New NoSQL Framework for Effective Interpretation in Web Usage Mining', *SSRN Electronic Journal*. doi: 10.2139/ssrn.3165272.

Menasce, D. A. (2002) 'Load testing of Web sites', *IEEE Internet Computing*, 6(4), pp. 70–74. doi: 10.1109/MIC.2002.1020328.

Mitchell, L.J. (2016)*Php web services, 2nd edition*. O'Reilly Media, inc. ISBN: 978-1491933091.

Parker, Z., Poe, S. and Vrbsky, S. V. (2013) 'Comparing NoSQL MongoDB to an SQL DB', in *Proceedings of the 51st ACM Southeast Conference. New York, NY, USA: Association for Computing Machinery* (ACMSE '13), pp. 1–6. doi: 10.1145/2498328.2500047.

Reddy, K. S., Reddy, M. K. and Sitaramulu, V. (2013) 'An effective data preprocessing method for Web Usage Mining', in *2013 International Conference on Information Communication and Embedded Systems (ICICES). 2013 International Conference on Information Communication and Embedded Systems (ICICES)*, pp. 7–10. doi: 10.1109/ICICES.2013.6508197.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B. & Wesslén, A. (2012) Experimentation in Software Engineering. Berlin Heidelberg: Springer-Verlag. ISBN 978-3642290435.

www.Mysql.com (hämtat 2020-12-09)

www.Mongodb.com/partnerss/opensource-technologies (hämtat 2020-12-09)

www.stackoverflow.com (hämtat 2021-04-03)

www.php.net (hämtat 2021-04-03)

www.Microservices.io (hämtat 2021-04-03)